NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA

THESIS

AUTONOMOUS AGENTS FOR DISTRIBUTED INTRUSION DETECTION IN A MULTI-HOST ENVIRONMENT

by

Dennis J. Ingram

September 1999

Thesis Advisor: Neil Rowe
Second Reader: Geoffrey Xie

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE September 1999	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE **Autonomous Agents for Distributed Intrusion Detection in a Multi-host Environment**	5. FUNDING NUMBERS
6. AUTHOR(S) Ingram, Dennis J.	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution unlimited.	12b. DISTRIBUTION CODE

13. **ABSTRACT *(maximum 200 words)***

Because computer security in today's networks is one of the fastest expanding areas of the computer industry, protecting resources from intruders is an arduous task that must be automated to be efficient and responsive. Most intrusion-detection systems currently rely on some type of centralized processing to analyze the data necessary to detect an intruder in real time. A centralized approach can be vulnerable to attack. If an intruder can disable the central detection system, then most, if not all, protection is subverted. The research presented here demonstrates that independent detection agents can be run in a distributed fashion, each operating mostly independent of the others, yet cooperating and communicating to provide a truly distributed detection mechanism without a single point of failure. The agents can run along with user and system software without noticeable consumption of system resources, and without generating an overwhelming amount of network traffic during an attack.

14. SUBJECT TERMS Intrusion Detection, Artificial Intelligence, Autonomous Agents, Computer Security	15. NUMBER OF PAGES 66
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

AUTONOMOUS AGENTS FOR DISTRIBUTED INTRUSION DETECTION IN A MULTI-HOST ENVIRONMENT

Dennis J. Ingram
Captain, United States Marine Corps
A.S., Southeastern Louisiana University, 1984
B.S., Park College, 1989

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 1999

ABSTRACT

Because computer security in today's networks is one of the fastest expanding areas of the computer industry, protecting resources from intruders is an arduous task that must be automated to be efficient and responsive. Most intrusion-detection systems currently rely on some type of centralized processing to analyze the data necessary to detect an intruder in real time. A centralized approach can be vulnerable to attack. If an intruder can disable the central detection system, then most, if not all, protection is subverted. The research presented here demonstrates that independent detection agents can be run in a distributed fashion, each operating mostly independent of the others, yet cooperating and communicating to provide a truly distributed detection mechanism without a single point of failure. The agents can run along with user and system software without noticeable consumption of system resources, and without generating an overwhelming amount of network traffic during an attack.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank my God for His assistance in this effort; my wife, Amy, for her understanding and support; and my kids, Rachel, Denny, and Hannah, for letting their dad finish his work.

I would like to thank my advisor, Professor Neil Rowe, for his assistance and advice on the project.

Special thanks to Professor Geoffrey Xie for providing the computers and resources necessary to complete this thesis.

Also, thanks to Major Jim Breitinger for his advice and for being a sounding board during the programming phase of the project.

I. INTRODUCTION

This thesis investigates using autonomous agents in a Windows NT network as intrusion-detection agents that act mostly independently yet share information. There are currently many intrusion-detection systems available, but most operate either on a separate computer monitoring the network traffic or as fully independent agents running on existing computers that report data to a central controller. My approach is to provide an agent that will run on some or all platforms in the network, and operate autonomously while at the same time cooperating with other agents to communicate threats throughout the network, to form a robust redundant detection system.

A. BACKGROUND

In the past decade, the Internet has grown from a fledgling network of computers to a multi million dollar industry. With everyone interconnected, the problem of security of information is a most definite concern. With all the information available on the Internet, intruders see the Internet as an easy opportunity for malicious mischief. Detecting an intruder in a network environment is hard for a human. The rates of data transfer and the amount of information flowing digitally through the physical media require an electronic means of surveillance. Even with a computer collecting the data, there is still too much information for a person to analyze and track in real time or even near-real time.

To protect important systems from hackers, intrusion-detection systems are becoming more prevalent. Hundreds of intrusion-detection systems now on the market claim to protect your system. Most of these systems rely on centralized control and centralized analysis of data to determine if an intruder has entered the system. This centralized scheme of control is vulnerable since an intruder can disable or bypass it by attacking just one host. Many of these systems are single-host-based systems that sit on the network and monitor all traffic flowing through a segment. Other systems have multiple agents that reside throughout the network on separate hosts and report any abnormalities or alerts to a central repository for logging and analysis. Some programs have tried to combat the centralization of control by allowing multiple controllers, but this is not common.

B. GOAL

My goal is to determine, through experimentation, if multiple autonomous intrusion detection agents can act mostly independently, running on many servers and workstations in the network, and can collaborate to form a network protection grid with no single controller or point of failure, without overburdening the network or an individual workstation with network traffic. Disabling any agent should alert other agents in the system that there is a potential problem.

C. THESIS ORGANIZATION

Chapter II describes background and puts this research into perspective. It discusses types of detection systems and requirements for a good intrusion detection system. Chapter III discusses other work in this area that attempts to solve the problem. Chapter IV describes my intrusion-detection agent, its components, and how each component operates. Chapter V provides details of experiments and tests run on the system and the results of these tests. Chapter VI provides conclusions and future work expectations.

II. INTRUSION DETECTION

A. BACKGROUND

Intrusion detection is an absolutely essential part of today's network-centric digital world. An intrusion into a computer system can be compared to a physical intrusion into a building by a thief: It is an entity gaining unauthorized access to resources. The unauthorized access is intended to steal or change information or to disrupt the valid use of the resource by an authorized user. Intrusion detection is the ability to determine that an intruder has gained, or is attempting to gain unauthorized access. An intrusion-detection system is a tool used to make this determination. The goal of any intrusion-detection system is to alert an authority of unauthorized access before the intruders can cause any damage or take any information, much like a burglar alarm system in a building. However, a digital computer system is far more vulnerable than a building and much harder to protect. The intruder can be hundreds of miles away when the attack is initiated, leaving behind very little evidence.

Intrusions generally fall into two categories: misuse and anomalies. Misuse attacks exploit some vulnerability in the system hardware or software to gain unauthorized access. Many of these attacks are well documented and are easily detected by computer systems, but new ones are constantly being discovered. Anomalies are harder to detect since they often originate from an inside user who already has access to the system. They are characterized by deviations from normal user behavior, and detection requires some type of user profiling to establish a normal behavior pattern.

B. TYPES OF DETECTION SYSTEMS

There are several types of detection systems on the commercial market. These systems can be used individually or can be combined to provide more protection.

1. Host-Based

A host-based system resides on a single host computer. It uses audit logs or network traffic records of a single host for processing and analysis. This type of system is limited in scope since it is only able to see its own host's environment, and cannot detect simultaneous attacks against multiple hosts.

2. Network-Based

A network-based system is a dedicated computer, or special hardware platform, with detection software installed. It is placed at a strategic point on a network (like a gateway or subnetwork) to analyze all network traffic on that particular segment. It can scan data traffic for known attack patterns. It can also determine Internet Protocol (IP) addresses that originate outside its subnet. This system can detect attacks against multiple hosts on a single subnet, but it usually cannot monitor multiple subnets at one time. It also cannot detect any host-based attack that does not pass through it.

3. Distributed

Distributed systems allow detection software modules to be placed throughout the network with a central controller collecting and analyzing the data from all the modules. This provides a robust mechanism for detecting intrusions across several subnets and several hosts. But it requires a dedicated computer to act as the central controller; centralization can make it vulnerable to attack.

C. CATEGORIES OF ATTACKS

Today's hackers use several categories of attacks ranging from simple to very complex. The basic categories are listed in Table 1.

One-to-one	Attacker uses a single machine to attack a single target machine. Example: sendmail bugs.
One-to-many	Attacker uses a single machine to attack many targets. Example: probes, denial of service attacks.
Many-to-one	Attacker divides assault among multiple outside machines to attack a single victim. This is difficult to detect because multiple connections from multiple sources look more innocent than multiple connections from a single source. Example: SYN flood using IP spoofing to deny services.
Many-to-many	Many collaborating attackers divide the tasks of probing/attacking multiple victims. This poses the same challenge as the "many-to-one" case with the added complexity of multiple target machines. This kind of attack is very difficult to detect. Example: "Smurf" attack from multiple sources.

Table 1: Attack Types (from DURST99)

D. REQUIREMENTS FOR A DETECTION SYSTEM

Joseph Barrus [BARRUS97] defined ten basic requirements for a good intrusion detection system:

1. A system must recognize any suspect activity or triggering event that could potentially be an attack.
2. Escalating behavior on the part of an intruder should be detected at the lowest level possible.
3. Components on various hosts must communicate with each other regarding level of alert and intrusions detected.
4. The system must respond appropriately to changing levels of alertness.

5. The detection system must have some manual control mechanisms to allow administrators to control various functions and alert levels of the system.
6. The system must be able to adapt to changing methods of attack.
7. The system must be able to handle multiple concurrent attacks.
8. The system must be scalable and easily expandable as the network changes.
9. The system must be resistant to compromise, able to protect itself from intrusion.
10. The system must be efficient and reliable.

E. SUMMARY

The problems facing computer administrators are enormous and still increasing in scope. Protecting computer systems from attack must be automated to be efficient. There are programs available that perform adequately, but all seem to have the disadvantage of either being single-host or distributed with a central analysis point, which are both vulnerable. A completely distributed system with independent agents, each performing its own analysis and still coordinating with all other agents, might be the best design for a modern network. There would be no single point of failure, and an intruder would have to disable or bypass all the running agents to succeed.

III. OTHER WORK IN THE AREA OF INTRUSION DETECTION

There are hundreds of systems available that perform intrusion detection, intrusion prevention, and system security checking. Many perform well and provide a robust detection mechanism, but few run in a fully distributed environment. Of those that are distributed, many are Unix-based systems and will not run on Windows NT platforms. There are fewer still that are portable between operating systems.

The systems most similar to the one presented in this thesis all have one major difference from it, in that they are hierarchical in nature. This places the highest vulnerabilities at the upper level of the hierarchy. Degrading or disabling a top-level monitor would severely limit the detection capability of the system. None of these systems mention the use of an alert level to determine if an attack is in progress.

A. ADAPTIVE INTRUSION DETECTION SYSTEM (AID)

AID [SOBIREY99] is a client-server architecture that consists of agents residing on network hosts and a central monitoring station. Information is collected by the agents and sent to the central monitor for processing and analysis. It currently has implemented 100 rules and can detect ten attack scenarios. The prototype monitor is capable of handling eight agents. This system currently runs only on UNIX-based systems.

B. AUTONOMOUS AGENTS FOR INTRUSION DETECTION (AAFID)

The AAFID architecture [ZAMBONI98] appears the most similar to the one I propose. AAFID is designed as a hierarchy of components with agents at the lowest level of the tree performing the most basic functions. The agents can be added, started, or stopped, depending on the needs of the system. AAFID agents detect basic operations and report to a transceiver, which performs some basic analysis on the data and sends commands to the agents. A transceiver may transmit data to a transceiver on another host. If any interesting activity takes place, it is reported up the hierarchy to a monitor. The monitor analyzes the data of many transceivers to detect intrusions in the network. A monitor may report information to a higher-level monitor. The AAFID monitors still provide a central failure point in the system. AAFID has been developed into two prototypes: AAFID, which had many hard-coded variables and used UDP as the inter-host communication, and AAFID2, which was developed completely in PERL and is more robust. They run only on Unix-based systems.

C. COMPUTER MISUSE DETECTION SYSTEM (CMDS™)

CMDS™ is a commercial product from Science Applications International Corporation (SAIC) [PROCTOR96]. It is a real-time audit reduction and alerting system that uses an expert system and statistical profiling to analyze audit records. The system uses distributed daemons running on host machines to monitor audit files. Information is sent to the CMDS central server for analysis by a rule-based expert system. It also uses a hierarchical architecture with several CMDS servers reporting to a higher CMDS system. It currently supports the operating systems SunOS, Windows NT, Solaris, Trusted Solaris, Ops Intel Workstation, Data General DSO, HP/UX, IBM LAN Server, Raptor Eagle Firewalls, ANS Interlock Firewalls, and SunOS BSM. This program appears to be robust across many platforms.

D. EVENT MONITORING ENABLING RESPONSE TO ANOMALOUS LIVE DISTURBANCES (EMERALD)

EMERALD [NEUMANN99] is a system developed by SRI International with research funding from DARPA. The EMERALD project will be the successor to Next-Generation Intrusion Detection Expert System (NIDES). It is designed to monitor large distributed networks with analysis and response units called monitors. Monitors are used sparingly throughout the domain to analyze network services. The information from these monitors is passed to other monitors that perform domain-wide correlation, obtaining a higher view of the network. These in turn report to higher-level enterprise monitors that analyze the entire network. EMERALD is a rule-based system. The target operating system has not been stated, but it is being designed as a multi-platform system. EMERALD provides a distributed architecture with no central controller or director; since the monitors are placed sparingly throughout the network, they could miss events happening on an unmonitored section. My approach is to employ agents on many hosts to attempt detection of all suspicious activity.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. HOST BASED AUTONOMOUS AGENTS

A. INTRODUCTION

Intrusion detection in a computer network is difficult. As detection mechanisms are designed and implemented, intruders discover new ways to infiltrate the system. Yet currently an intrusion detection system is the best way to protect your system from intruders.

In designing my approach, I concentrated on agent communication and coordination. I did not try to incorporate extensive detection mechanisms into the code, as my goal was to determine decentralized agents could be designed to run and communicate without interfering with normal network and CPU operation. A related thesis by Stephen Kremer [KREMER99] deals with a broader range of intrusion and misuse phenomena in a network.

B. SOFTWARE CHOICES

The agent base design is implemented in Java® version 1.1.8 to enable it to be platform independent. Initial tests of the communications mechanisms were done on Windows NT 4.0 workstations and Windows NT 4.0 server and Linux version 5.2. Several trial runs were also conducted in a mixed environment with NT 4.0 workstation, NT Server, and Linux 5.2, running together. One problem discovered was that platform independence was difficult to achieve for an intrusion-detection system because many of the mechanisms used to detect intrusions, such as system logs and system alert facilities, are specific to a certain platform. For example, a collection of data from a system log is processed differently on a Windows NT platform than it is on a Linux or Sun® workstation. For this reason, I chose Windows NT as the single platform for final testing.

C. AGENT DESCRIPTION

I chose the name IDAgent because the agents were being designed for "Intrusion Detection" and also because the agents were supposed to operate "InDependently".

Figure 1 shows a simple block diagram of the IDAgent components needed for a single host. All major components of the agent are constructed as threads to allow them to run concurrently. The main components and data structures are: Controller module, TCP Receiver, UDP Receiver, TCP Transmitter, UDP Transmitter, Agent Window Manager, Host Sensor, Log Sensor, Message class, and Contact List of known agents.

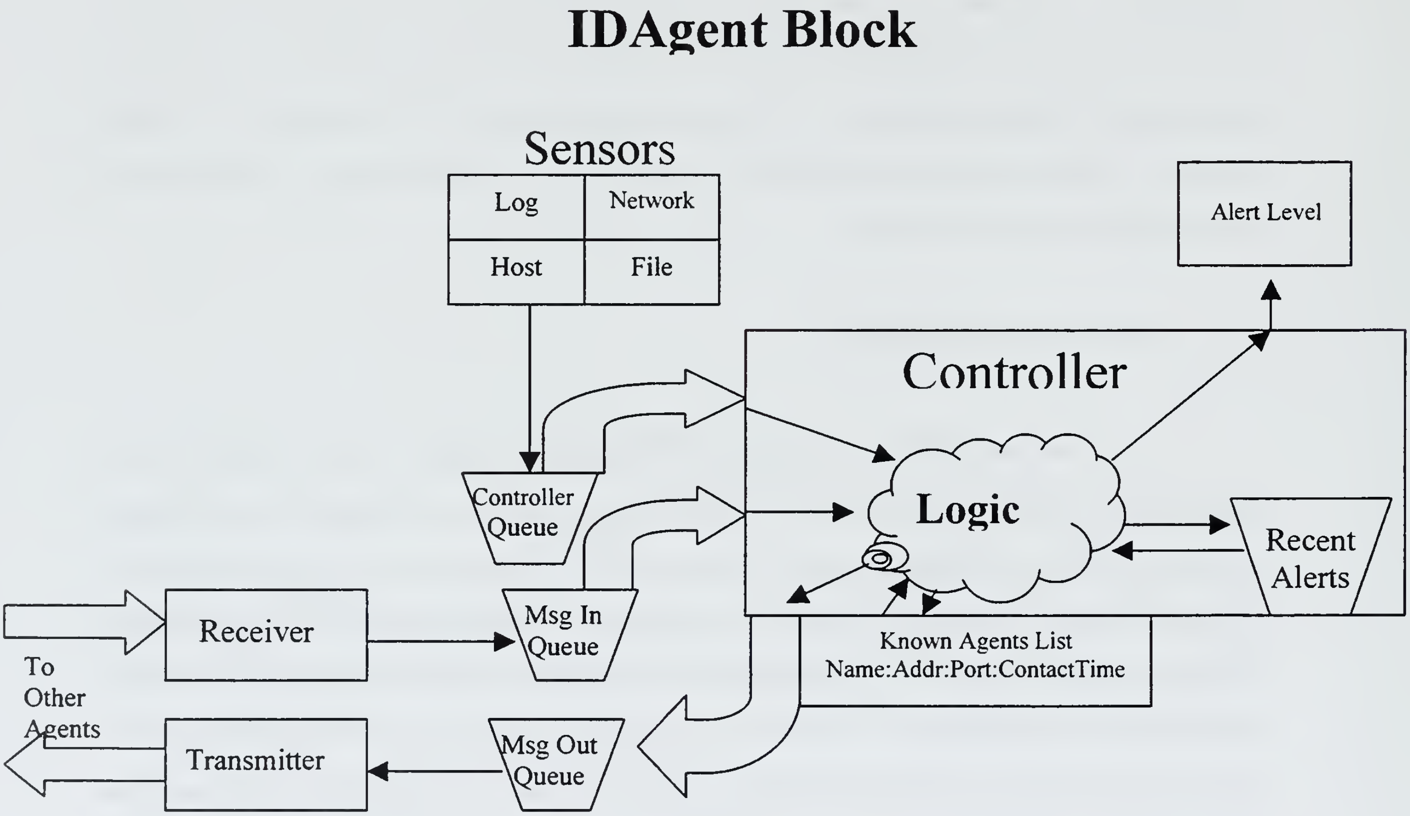


Figure 1: IDAgent Block Diagram

1. Agent Window Manager

The Agent Window Manager was written initially by Major Jim Breitinger to give a basic graphical window to display information for another project. I modified it to be used as a user interface with the IDAgent. It contains a display frame of 500 by 320 pixels for a text display area. The lower portion of the window contains an area with control buttons on a colored background of green, yellow, or red, depending on the current alert level of the IDAgent. The control buttons allow the generation of debug data for testing, the display of the current contact list of known agents, a display of alert messages that caused a change to the alert status, and a status which indicates a numerical value of the current alert level.

2. Controller Module

The controller is the brain behind the IDAgent. Once the main program initializes all variables and the Window manager is started, the Controller thread is started. The controller, as with almost all other threads, is in a suspended mode when there is no activity. Any activity in any sensor, receiver, transmitter, or Window Manager of the IDAgent will activate the controller to analyze the activity. Its primary purpose is to analyze incoming messages from other agents and internal sensors and determine if an intrusion is in progress. The controller updates the Alert status of the agent depending on the messages it receives. The Alert status can range from 0.0 to 1.0 and represents the likelihood that an intrusion is taking place. Alert levels of 0.0 to 0.4 will display a green indicator in the user display; 0.4 to 0.7 will show a yellow indicator; and above 0.7 will display red.

The Alert level is increased depending on the "weight" of the message. All messages are initially sent with a weight value of 0.0; this prevents the message from affecting the alert level until the controller has analyzed the message. Once analyzed the message will either be saved for future reference or the weight and alert level will increase. Each message is analyzed when it arrives. If an attack is suspected, then an appropriate weight value is assigned to that message, which in turn increases the alert level of the IDAgent. If the message is not considered an attack, then the message weight remains 0.0.

The normalized increase in the alert level, as shown in Figure 2, is inverse-exponential as the alert value increases. $N = ((1.0 - A) * W) + A)$, where A = the current alert level, W = the weight value of a message, and N = the new alert value. The alert level will approach 1.0 if many alerts are received. If no alerts are received within two transmit intervals (10 minutes in the current implementation), the alert level will decrease following a negative exponential curve, i.e., N = (A * Degradation Factor) where degradation factor is a fraction (0.9 in the current implementation). Figure 3 shows the decrease of alert level over time if no alerts are received. Eventually, the alert level will approach 0.0.

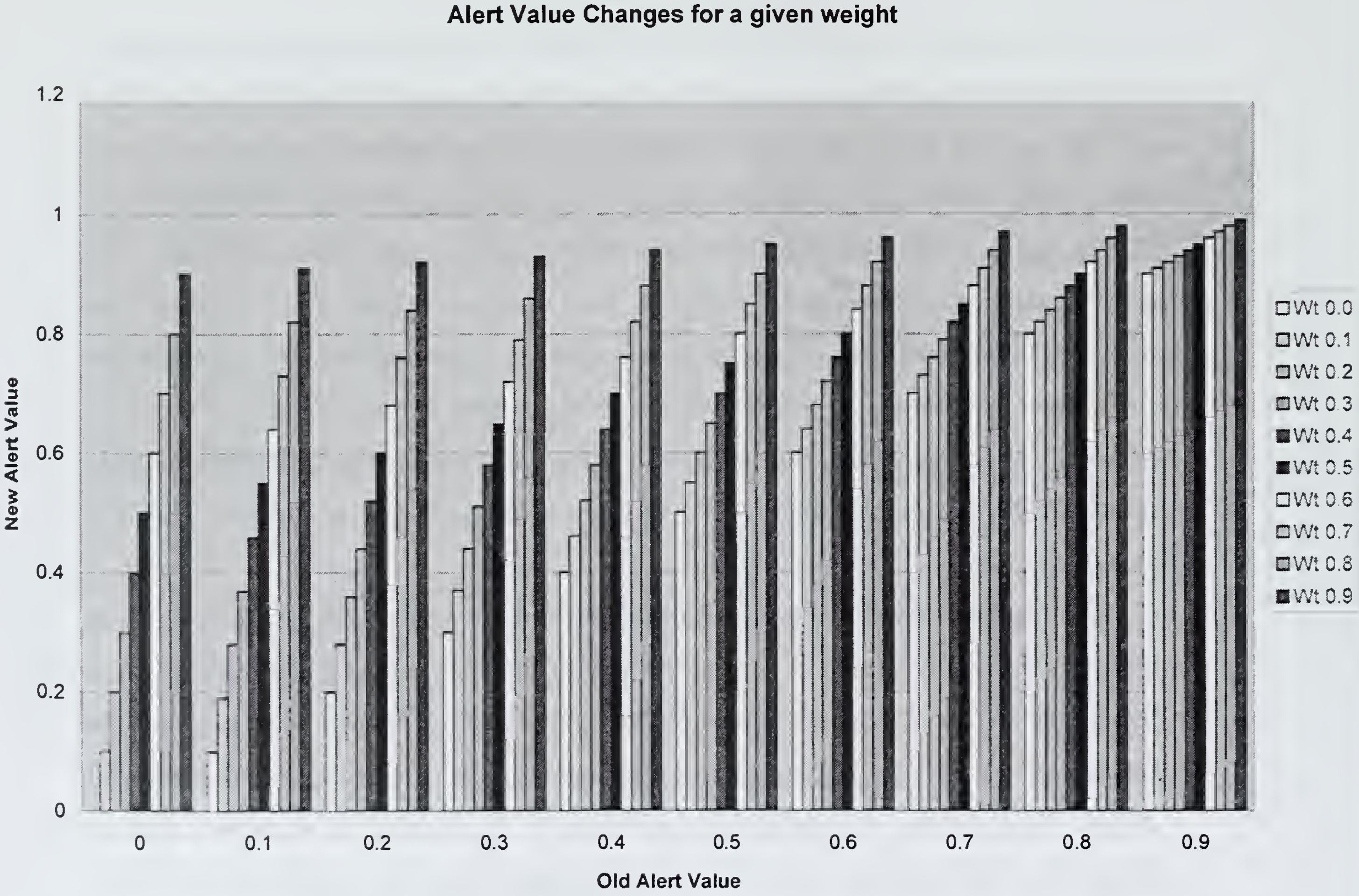


Figure 2: Alert-Level Increase

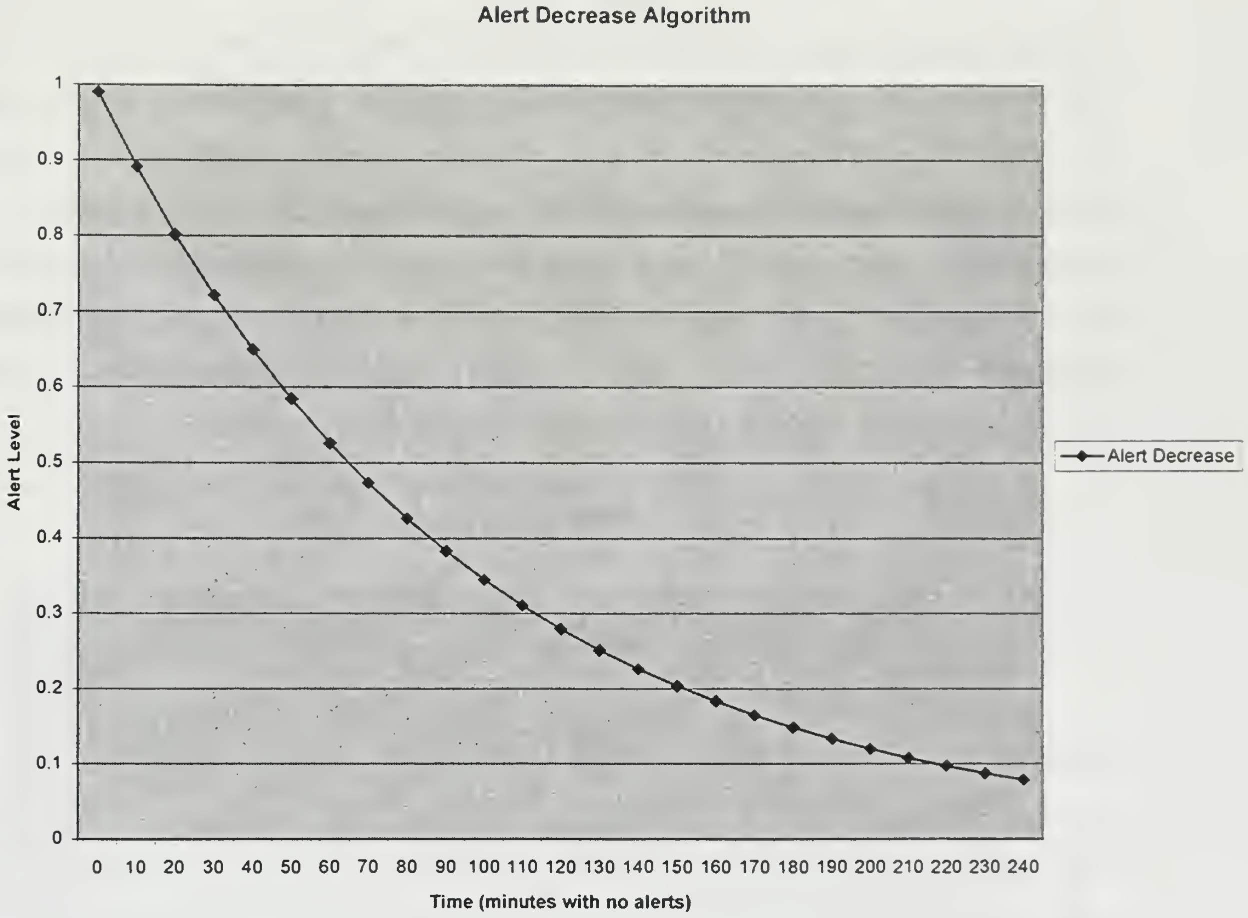


Figure 3: Alert-Level Decrease

If important information is received from an internal sensor, its agent's controller will construct a message and send it to other agents in the network to notify them of an event or action that is taking place on its own host. Messages from another agent are not forwarded to other agents in the network to prevent duplicate message traffic. For an example, assume in a network of twenty computers that the agent on computer nine detects a failed login attempt. Its controller analyzes the attempt and constructs an alert message that is sent to all nineteen other computers. Now should the person attempt a login on another computer, it too would be detected and sent to all agents.

The current implementation includes only basic intrusion capabilities for testing. It includes an external sensor written by Stephen Kremer [Kremer99] that scans the system logs for login attempts. The log sensor (section 11) is the internal interface between his program and the IDAgent that retrieves the login attempts. The attempts are passed to the agent where they are processed by the controller. The controller analyzes each failed login attempt and calculates a fraction of failed attempts as compared to the total attempts. This calculation is done both for attempts on a single host and the network as a whole. If either the host or network fraction reaches the threshold value for the agent, an alert message is constructed with a weight value of (fraction – threshold). To overcome the problem that occurs with small login ratios (1 login attempt and 1 failure is 100% failure rate), the following formula is used to calculate the login-failure fraction: Fraction = (LoginFailures – 1 / TotalAttempts). Figure 4 shows the effect of this calculation.

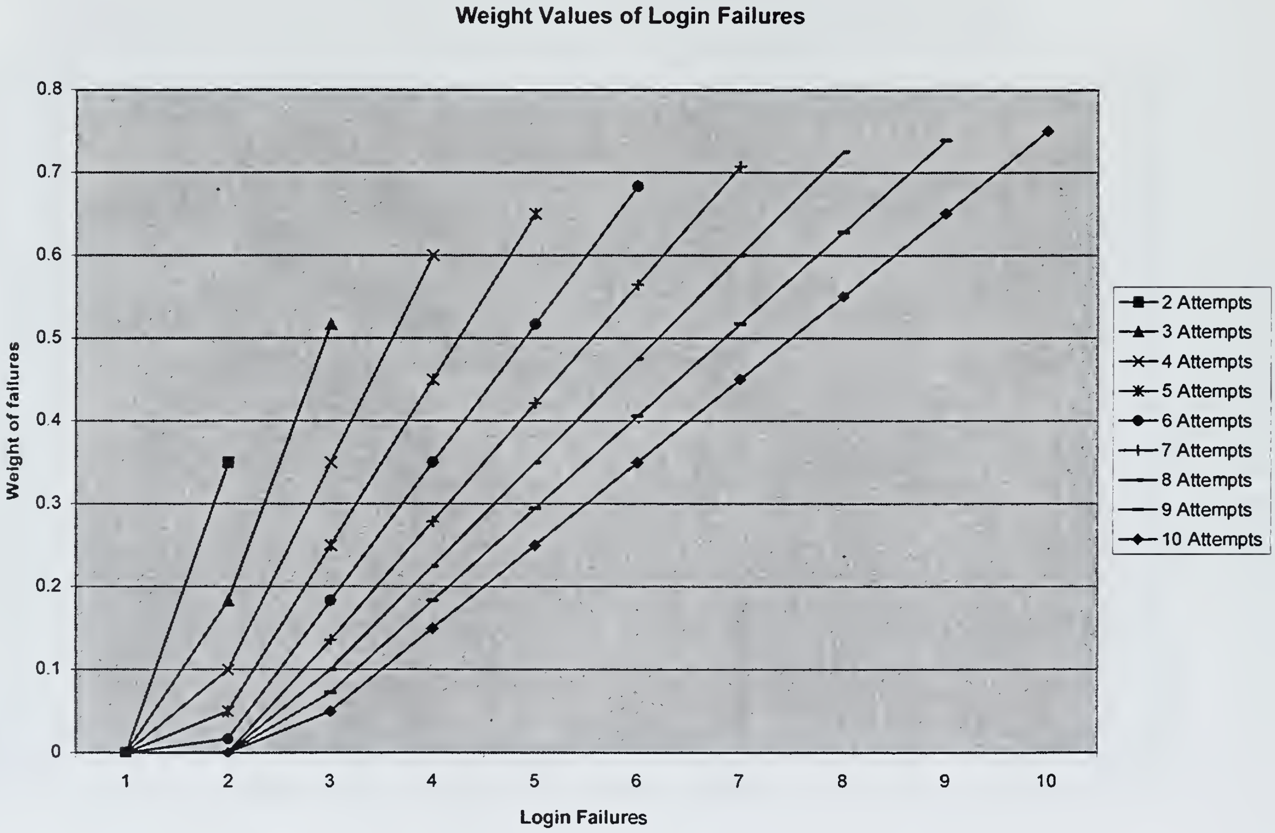


Figure 4: Login Failure Calculation

Another internal sensor included in the agent is the host sensor, which uses the contact list of known agents to determine if an agent has stopped responding. The host sensor monitors how many remote agents have contacted it and checks to make sure they are all still functioning (see section 9). If the number of agents not responding reaches a threshold, a message is sent to the controller.

3. UDP Transmitter

The Unreliable Datagram Protocol (UDP) Transmitter thread generates a UDP packet that contains the port number that the agent's Transmission Control Protocol (TCP) receiver is listening on and identification of the local host on which it is running. It sends this packet as a broadcast message to the network on a port number determined at startup. This broadcast occurs every five minutes. The thread initiates the first contact and maintains the contact between all agents.

4. UDP Receiver

The UDP Receiver's primary function is to receive the UDP broadcast messages of other agents and process them. The broadcast port number by default is 8000. Although any unused port number may be designated at system startup, all agents in the network must be running on the same port number. The UDP Receiver establishes a listener on the given port and waits for a broadcast message from another IDAgent. If a message is not in the correct format, an exception is generated and the broadcast is discarded. Otherwise, a contact record is created with the remote agent's identifying information, the port number of its TCP receiver, and the time that the message was received. The contact record is placed in a list of known contacts that is used by the controller and transmitter when sending messages. Each time a broadcast is received, the new contact record is compared to the contact list. If a match is found, the timestamp and the TCP receiver port number of the original record are updated. The port number is updated in case an agent was restarted and is now listening on a different TCP port. The timestamp allows the controller to determine the last time that an agent contacted it to aid the host sensor in detecting a non-responding host. If an agent fails to broadcast for 3.5

transmit intervals, it is considered by other hosts to be non-responding and this may result in an alert being generated (see section 9).

5. TCP Transmitter

The Transmission Control Protocol (TCP) Transmitter thread sends messages between agents. A message contains information that an agent needs to report an attack. Since the delivery of such a message helps in the detection of an intruder, some guarantee of delivery must be expected. The Unreliable Datagram Protocol (UDP) Transmitter as the name implies does not provide such assurance, but the TCP protocol is connection-oriented and does [COURTOIS98]. Message composition is covered in more detail in section 7. In the current configuration, the transmitter will deliver any message to all known agents on the contact list. It establishes a connection with the remote agent's TCP Receiver, transmits all currently available messages, and closes the connection.

6. TCP Receiver

The TCP Receiver thread picks a port to listen on that is not being used by any other components of the computer on which the agent resides. It returns this port to a global variable in the IDAgent so the UDP transmitter explained above will be able to access it to tell other agents which port the receiver is listening on. When a message is received, the TCP Receiver queues it for the controller in the message-in queue and continues listening for additional messages. A TCP socket connection must be established between two agents for the message transfer to take place. If a connection cannot be established, the sending agent should become aware of the problem and can report it to its own controller.

7. Message Class Data Structure

A Message Class defines message objects that can be constructed and transmitted from one host to another. The class contains a message code, a data field for a description of the message, an identifier, a target address, a source address, a time stamp, and a message weight. The message code indicates why the message was sent. The string data field relates to the code and provides additional description of the code. The identifier supplies operands, if any, for the code. For example, if a message were for a failed login attempt, the identifier would store the account name. The target address is the Internet address and host name of the recipient. The source address is that of the current host. Each message is given a timestamp at origination. The message weight is the relative importance of the message as determined by the controller following the methods in section 2. In the current configuration of the IDAgent, the message size is 787 bytes when the host name is eight characters.

8. ContactList Class Data Structure

The ContactList class is a data structure storing information about other known agents. It consists of an InetAddress, a port number, and a timestamp. The InetAddress is a Java data type that contains the Internet address and host name of a remote host. The port number is the port that the remote host has a TCP receiver listening on. The timestamp contains the last contact time of a remote agent.

9. Host Sensor

The function of the host-sensor thread is to determine if any remote agents are not broadcasting using the UDP transmitter. It checks the contact list of known agents and compares the time of last contact to the current time. If the host has not responded, an alert message is generated and placed in the controller queue (see Figure 1). The controller will calculate a message weight based on previous messages. It will then use the message weight, of this internally generated message, to determine if there is sufficient evidence to update the system alert level. The fraction of hosts not currently responding determines the weight, to limit false alerts when an agent is stopped or restarted by an administrator. The host sensor does not cause any external messages to be generated. It is assumed that each IDAgent will detect a non-responding host, and therefore external messages would be redundant.

10. Alert Parser

The alert parser thread is a utility thread that maintains the internal messages that the controller uses to detect intrusions. The alert parser runs approximately every thirty minutes or six broadcast intervals. It looks through a list of old alerts and discards any over twenty-four hours old. It scans a list of recent alerts and places any over twelve hours old into the old alert list. This allows the controller to run more efficiently when it only needs to scan recent events. For the current configuration, only the recent alerts are used for processing. The alerts over twelve hours old were included for future sensor capabilities and are not currently used.

11. Log Sensor

The log sensor is another independent sensor thread. LT Steven Kremer [Kremer99] wrote the log sensor that automatically retrieves all login attempts from the system log and passes them to the internal log sensor thread. This requires that the system audit capabilities be turned on. Once the log sensor is instantiated, it periodically checks to see if anything has been passed in. If a login was attempted, a message to the controller is generated indicating the time, type of attempt, the host that the attempt was made on, and the name of the account used for the attempt. The controller stores the message and analyzes all previous attempts to try to detect a pattern. The log sensor currently only detects login attempts, but it could be modified to detect other system events. The log sensor does not perform calculations to adjust the weight of the message.

D. SUMMARY

The IDAgent has been designed in a modular fashion to allow easy incorporation of new functionality and changes to internal components. It contains transmitter and receiver components for the transmission of messages to other agents in the network and a controller component to analyze the messages received. Other threads provide rudimentary intrusion-detection facilities and permit analysis of the data traffic and functioning of the components.

THIS PAGE INTENTIONALLY LEFT BLANK

V. EXPERIMENTS

A. INTRODUCTION

To conduct experiments, a program was designed that uses some simple detection mechanisms along with all the necessary components to transmit and receive data over the Internet. The first test was performed in the early stages of code development. Only a basic agent skeleton with TCP Transmitter and TCP Receiver classes was used in order to make an initial determination of network overhead usage. Follow-on tests were then conducted with other parts of the agent operating to get the full effect on network and CPU utilization. Finally, simulated alert messages were sent to see how the agents reacted and what impact this reaction would have on CPU utilization of the host.

B. RESTRICTED NETWORK TESTING

Initial throughput testing of the IDAgent was conducted on a closed network of three Micron 166Mhz Pentium computers, each running the Windows NT 4.0 operating system as server or workstation. Two of the machines were configured as workstations with 32MB of RAM, and one was configured as a server with 64MB of RAM. To prove portability of the basic agent, tests were also performed on the same machines running the Linux operating system version 5.2. However, no other portions of the testing were done on Linux, and the results were only used to show portability of the agent to other platforms.

The agents had no detection capabilities or message processing capabilities during the initial testing. Only the network-bandwidth utilization was compared. Using an Observer® network-packet "sniffer" (a software program used to capture and analyze information being transmitted over a network), I monitored the network to determine the average bandwidth utilization for the three agents on a 10Mbps Ethernet 10BaseT network. The bandwidth measurement includes usage resulting from network polling, broadcasts, and network overhead on both the Windows NT and Linux operating systems. The IDAgents were configured to send 5,000 static messages of approximately

155 bytes each to each of the other agents. With three agents running, 45,000 messages or approximately 6.975 Megabytes of data was transmitted. The test was repeated three times to get an average transmit time. Figure 5 shows the results.

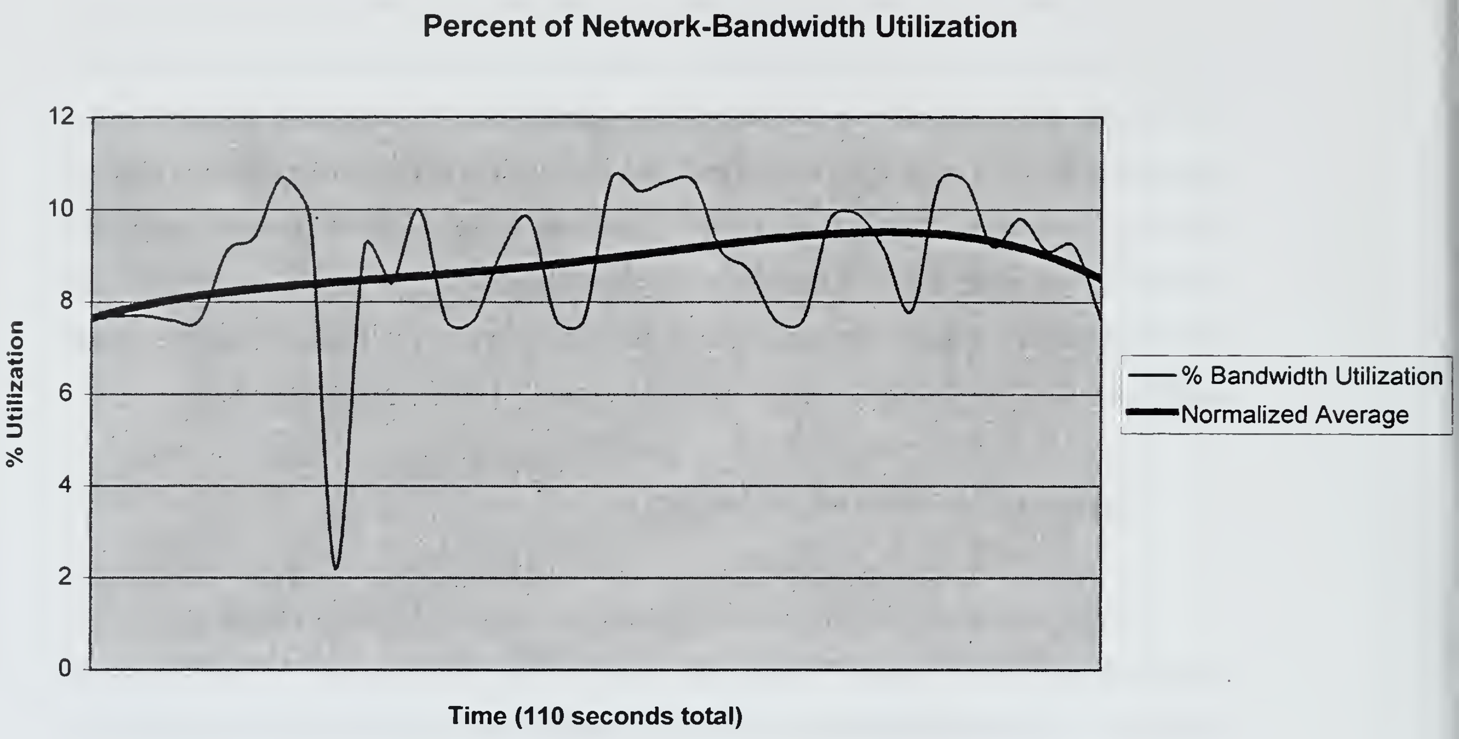


Figure 5: Network Bandwidth Utilization

The transmission of 45,000 messages took approximately 110 seconds, and the average bandwidth never exceeded 10% of the 10Megabit Ethernet network. The results were very encouraging since it will rarely be expected that an agent will need to transmit 5000 messages in such a short time.

C. GENERAL NETWORK TESTING

The second test was conducted on an open network in the Computer Science Department at the Naval Postgraduate School. The subnet I used is the same one used by most students, faculty, and researchers in the department. I used the same three computers and added four additional workstations, all running the Windows NT operating system version 4.0 workstation. The IDAgent was fully configured and included login detection, as described in Chapter IV section C sub-section 11, and host failure detection as described Chapter IV section C sub-section 9. Any successful or unsuccessful login attempts generate a message from the agent sent to all other hosts that have the IDAgent running; a broadcast message is sent by each host at five-minute intervals to update the contact list of known agents. Some general assumptions were made for this test to determine what an adequate number of login attempts should be. We estimated the number of daily login alerts based on a ten-user network. We assumed each user performs a login approximately three times a day. We assumed each user locks the computer screen an additional four times a day, requiring a password to unlock it, and generating an authentication alert. Windows NT authenticates users on the network who map a drive to a shared resource, which also generates an authentication alert on login since the resource is still open during a screen lock. It is assumed for this scenario that each user maps two network drives: one for shared applications and one for a shared file storage location. An expected login failure rate of 15% is set as the threshold in the IDAgent to reduce false alerts. With these assumptions, a network of ten users will generate approximately 130 login alerts per day, which will average approximately 16.25 logins per hour. Table 2 shows the expected login alerts and the acceptable login failure rate for other numbers of users.

Users	Logins /Day	Locks/Day	Mapped Drives	Estimated Alerts	Acceptable Failure Rate
10	3	4	2	130	19.5
20	3	4	2	260	39
30	3	4	2	390	58.5
40	3	4	2	520	78
50	3	4	2	650	97.5
100	3	4	2	1300	195

Table 2: Estimated Login Alerts

Using the same network sniffer as in earlier testing, I monitored the network with no agents running for one hour to establish baseline utilization. I then ran seven IDAgents on the network for one hour, generating over 50 login alerts and producing over 400 message transmissions. This is three times more than the average number for an hourly period in my assumptions.

Figures 6 and 7 show the average number of network packets per second, average number of broadcasts per second, and percentage of network-bandwidth utilization for the one hour period both with and without agents running. The average number of packets sent while the agents were running was actually slightly lower than without. The average number of broadcast messages increased slightly as expected; the average network utilization decreased slightly as shown in Figure 7, which was not expected. However, looking at the percentage of bandwidth used, the slight drop is insignificant when compared to the total bandwidth available. The "average maximum utilization" averages the peak bandwidth usage for each ten-second interval. This average went up slightly from 1.9 to 2.3, which indicates that the packet transmissions show more short bursts of data. From the data collected, it appears that the IDAgent has little effect on bandwidth consumption in an open network. During the one-hour time that the agents were running, approximately 64,000 packets were captured. Of those packets, only 5,391 were from one of the computers running an IDAgent, which is about 8%. The remaining 92% were from normal network activity.

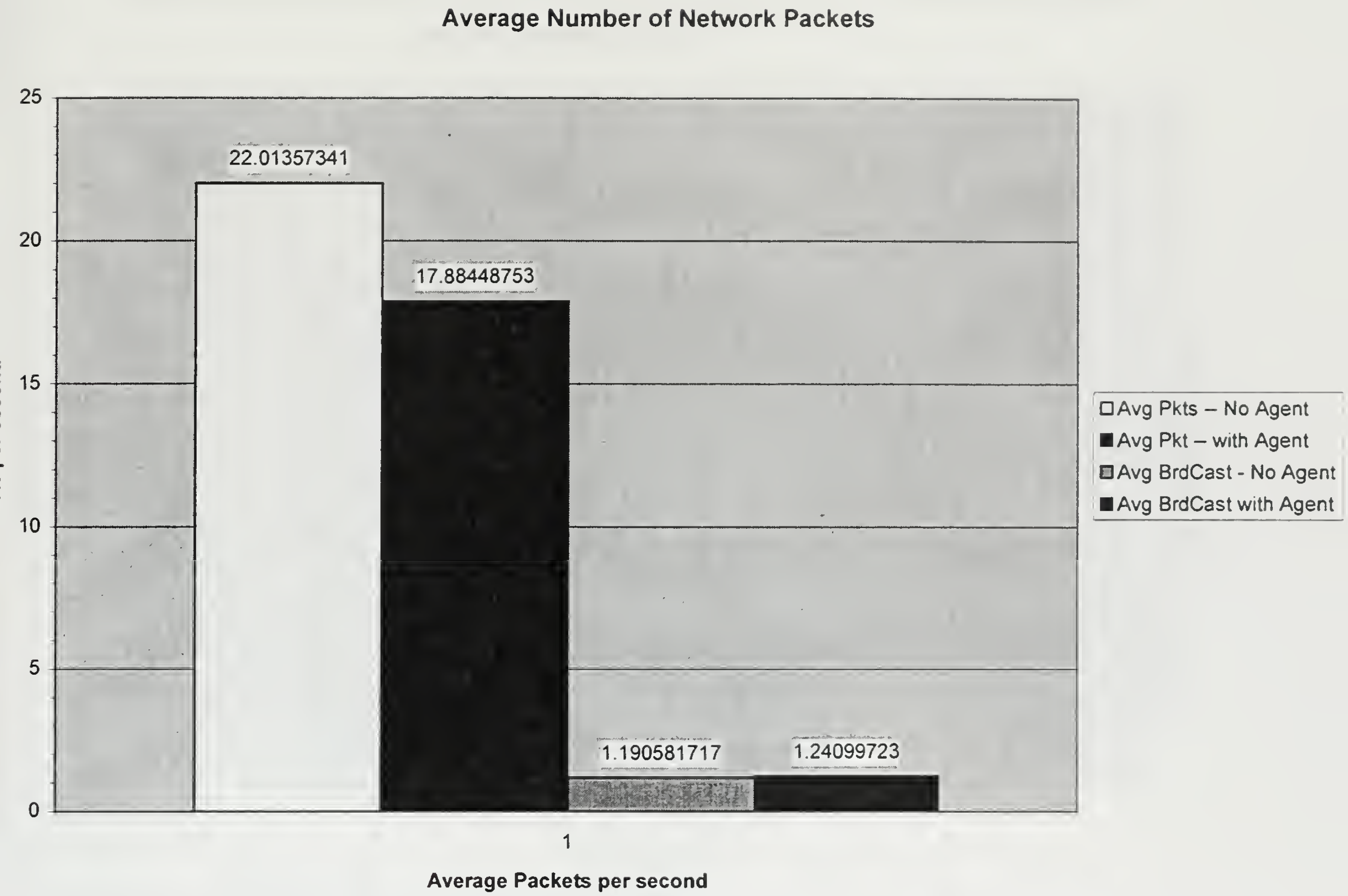


Figure 6: Average Packets and Broadcasts

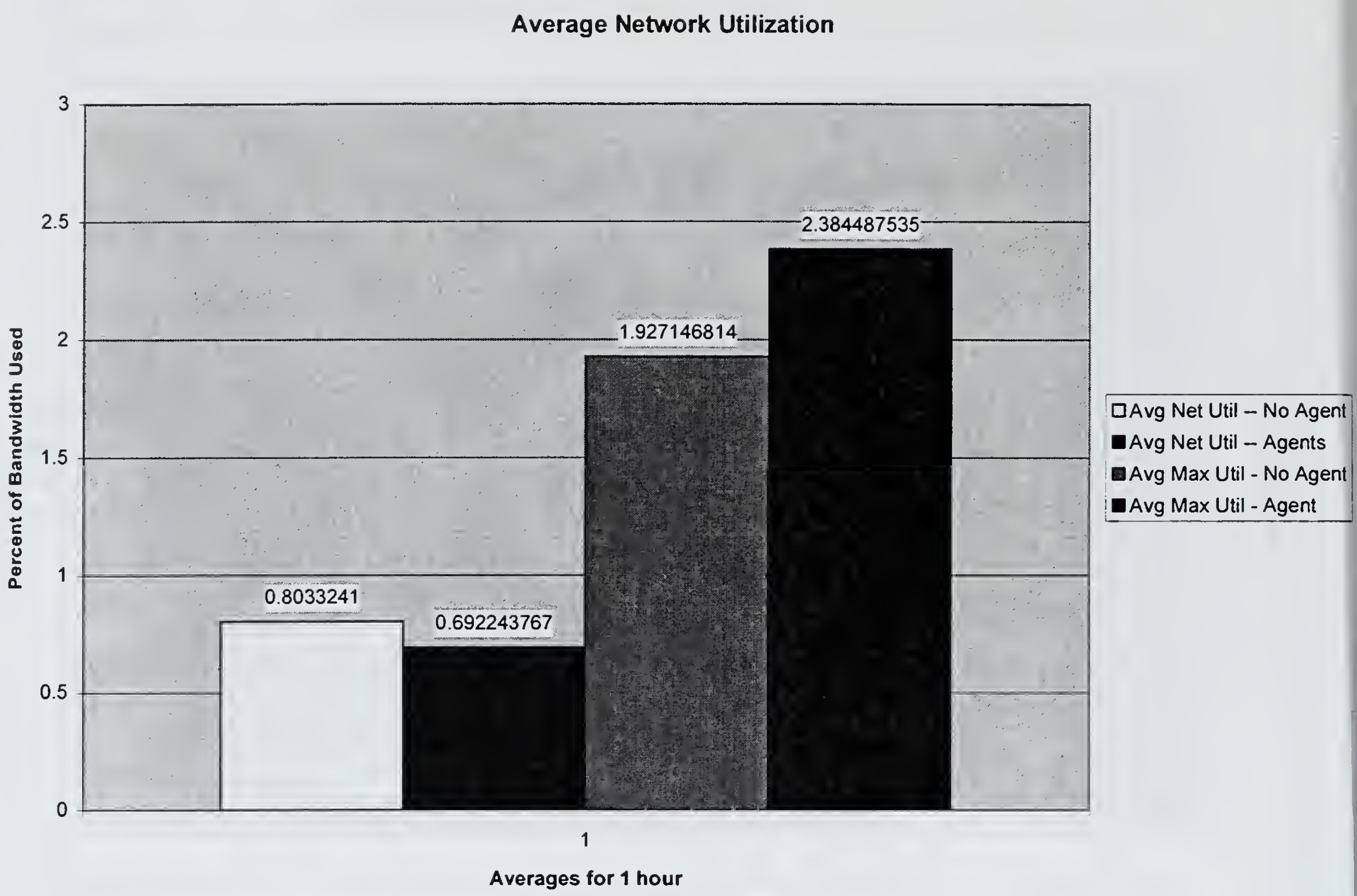
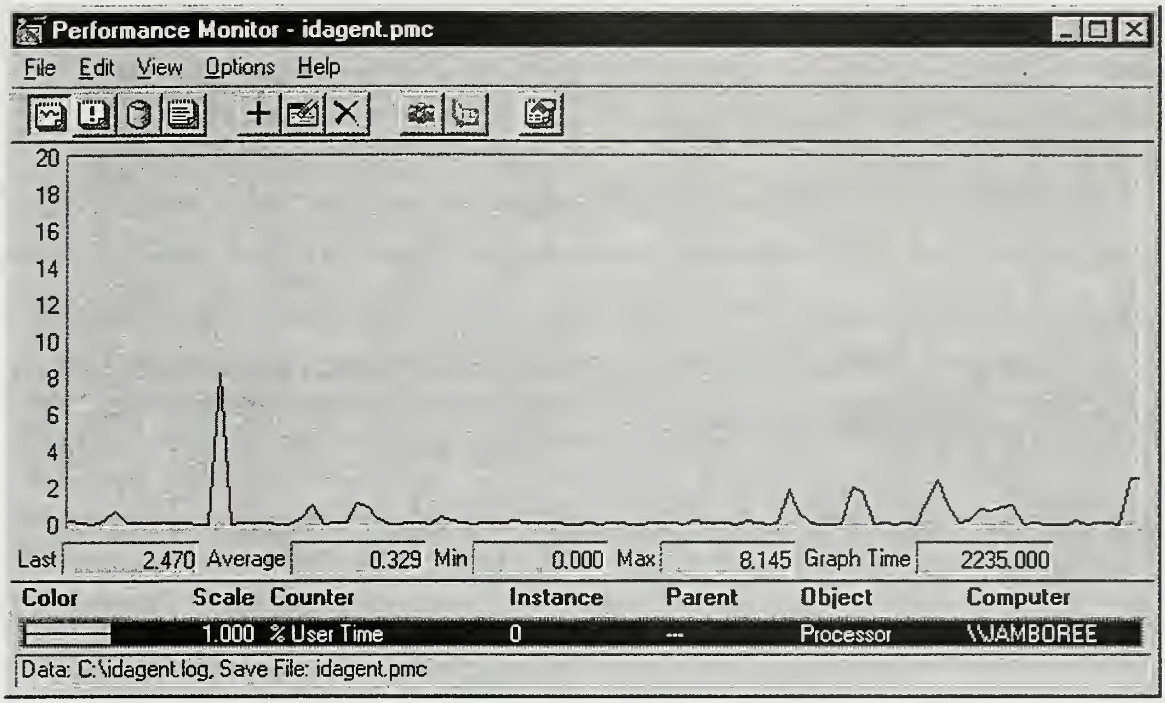


Figure 7: Percent Network Bandwidth Utilization

D. CPU UTILIZATION TESTS

Another test was conducted using the Windows NT performance monitor and logging tool. I was able to log and graph the CPU utilization over time with an IDAgent running to see its impact. I configured the performance monitor to log processor usage for user programs and started one IDAgent; no other user programs were running on its computer. An IDAgent was also started on another host and login alerts were generated from both computers. During the thirty-minute analysis period, approximately 20 alerts were generated. Figure 8 shows that the maximum CPU utilization of the agent was

8.145%. The average utilization over the entire period was 0.329%. There are several small usage periods, when the IDAgent was active in receiving and sending messages.


Performance Monitor - idagent.pmc
File Edit View Options Help
20
18
16
14
12
10
8
6
4
2
0

Last 2.470 Average 0.329 Min 0.000 Max 8.145 Graph Time 2235.000

Color	Scale	Counter	Instance	Parent	Object	Computer
▬	1.000	% User Time	0	---	Processor	\\JAMBOREE

Data: C:\idagent.log, Save File: idagent.pmc

Figure 8: CPU Utilization

E. SIMULATED ATTACK SCENARIOS

Several scenarios were used to test the reaction of the IDAgent. Three computers were used. In the first scenario, all three computers had several successful logins from users; then one computer had a series of unsuccessful login attempts on a single account. In the second scenario, several successful login attempts were performed on each computer, followed by a series of unsuccessful attempts. In the third scenario, all three computers were used, and many rapid consecutive unsuccessful login attempts were made from a single administrator account on one machine.

1. Single-Target Attack

After allowing all three agents to run for several minutes with no activity, two successful logins were made on each computer followed by an attack on machine three. The attacker produced six successive login failures. Table 3 shows the login attempts and reactions of the agents with their corresponding alert level changes. Machine three responded differently because its weight calculation was based on attempts being made on its own host, while the other two machine calculations were based on attempts throughout the entire network because the messages originated from another machine (See Chapter IV, section 2 for calculation details). The result is a higher alert level on the machine where the attack is taking place.

Total # of Attempts	Total # of Failures	Machine #1 Message Weight	Machine #1 Alert level	Machine #2 Message Weight	Machine #2 Alert level	Machine #3 Message Weight	Machine #3 Alert level
7	1	0.0	0.1	0.0	0.1	0.0	0.1
8	2	0.0	0.1	0.0	0.1	0.1	0.19
9	3	0.072	0.1648	0.072	0.1648	0.249	0.392
10	4	0.150	0.290	0.150	0.290	0.35	0.605
11	5	0.2136	0.4417	0.2136	0.4417	0.4214	0.771
12	6	0.2666	0.5905	0.2666	0.5905	0.475	0.880

Table 3: Alert Levels for Single Target Attack

2. Multiple-Target Attack

The second scenario was much like the first but with an attacker attempting to login on to all three machines simultaneously instead of just one. Table 4 shows the results of the test. The alert levels for all machines were very close together since login failures were spread across all hosts. The second machine reached a yellow alert level of 0.423 on the twenty-first login attempt with three local failed attempts, four remote failed attempts, and fourteen successful logins. The remaining machines reached a yellow alert level of 0.486 after two more attempts; one successful and one failure. A total of twenty-three logins were attempted; eight login failures and fifteen successful logins.

Total Login Attempts	Machine #1 Login Failures	Machine #2 Login Failures	Machine #3 Login Failures	Machine #1 Message Weight	Machine #1 Alert Level	Machine #2 Message Weight	Machine #2 Alert Level	Machine #3 Message Weight	Machine #3 Alert Level
11	1	0	0	0.0	0.1	0.0	0.1	0.0	0.1
13	1	1	0	0.0	0.1	0.0	0.1	0.0	0.1
14	1	2	0	0.0	0.1	0.05	0.1450	0.0	0.1
16	1	2	1	0.0375	0.1337	0.0375	0.177	0.0375	0.1337
17	1	2	2	0.0852	0.2076	0.0852	0.247	0.0852	0.2076
19	2	2	2	0.1131	0.2972	0.1131	0.3324	0.1131	0.2972
21	2	3	2	0.1357	0.393	0.1357	0.423	0.1357	0.393
23	2	3	3	0.1543	0.486	0.1543	0.512	0.1543	0.486

Table 4: Alert Levels for Multiple Target Attack

3. Network Saturation Test

The IDAgent is designed to suspend transmission of messages for a short period of time if it comes under a repeated attack, to prevent a flood of network traffic from its own messages. To test this, three test machines were started and 40 rapid login attempts were made against an administrator account on a single host. After transmitting 25 messages to the other agents, the IDAgent being attacked continued to log the attack, but it did not continue transmitting messages until five minutes after the attack had stopped. Agent response was successful: The attacked machine had an alert level of 1.0, the highest that can be reached, while both remaining agents had an alert level of 0.999.

F. SUMMARY

Testing showed that neither CPU utilization nor network utilization were heavily loaded by the IDAgent. Even with over 50 login attempts within one hour, the network traffic, broadcasts, and processing did not interfere with normal computer and network operations. The IDAgent was also able to detect several scenarios of login attempts from both a single host and multiple hosts, and escalated the alert level of each agent appropriately.

VI. CONCLUSION AND FUTURE WORK

A. CONCLUSION

This thesis has proposed distributed nonhierarchical autonomous agents as an intrusion-detection mechanism. Testing demonstrated that such use of an agent in this environment can be successful.

B. REQUIREMENTS REVIEW

We can assess our system in terms of the ten basic requirements for a good intrusion detection system listed in Chapter II:

1. The System Must Recognize Suspect Activity of a Potential Attack

The prototype system could effectively recognize failed logins, both on a single host and across distributed hosts. To recognize other types of activity, sensors would have to be written. The modular design of the IDAgent allows the straightforward integration of new sensors.

2. Escalating Behavior Should Be Detected at the Lowest Level Possible

The requirement to detect an intruder at the lowest level possible is very subjective. Triggering an alert the instant a failed login occurs would generate a large number of false positive alerts; waiting until an attack is absolutely certain might be too late. The threshold values in the IDAgent allow the level of detection to be adjusted to meet requirements. I believe my IDAgent detected login attacks at an appropriate level.

3. There Must Be Inter-host Communication Regarding Intrusions and Alert Levels

The IDAgent program was designed specifically to meet this requirement. Its transmitter and receiver components are the means of communication, and the Message Class data structure carries the information between hosts.

4. There Must Be Appropriate Response to Changing Alert Levels

This requirement was not implemented in the current configuration of the IDAgent.

5. The System Must Incorporate Manual Control Mechanisms for Administrators

The user interface for the IDAgent includes some control for debugging and determining the status of the agent. There are no controls for resetting thresholds or other parameters, but they could easily be added.

6. The System Must Be Adaptable to Changing Methods of Attack

This requirement was only partially met because only login sensors were written. Multiple sensors would be needed to detect changing attack methods.

7. The System Must Be Able to Handle Multiple Concurrent Attack Threads

IDAgent is a multi-threaded application that is capable of detecting multiple attack scenarios. If multiple login attacks were taking place, the IDAgent should be able to detect all suspicious activity.

8. The System Must Be Scalable and Easily Expandable

This requirement is fully met by IDAgent. To scale to a large network, you simply start agents on the added hosts. Expandability is allowed through the modular design of the agent.

9. The System Must Be Resistant to Compromise and Able to Protect Itself from Intrusion

This is left as future work. Java® provides many built-in security features, though none were incorporated yet.

10. The System Must Be Efficient and Reliable

Determination of efficiency was one of the primary goals of this thesis and has been adequately achieved in this prototype. Network bandwidth consumption and CPU utilization were both tested. The system was reliable under our limited testing.

C. FUTURE WORK

Some of the following would provide for a more robust agent for future work and testing:

1. Secure Message Transfer

The current agent does not incorporate security or secure message handling to prevent blocking of messages or generation of false messages. Java® does provide built-in encryption mechanisms that could be used.

2. Agent Authentication

How does one determine if an agent that is responding is really a trusted agent or a piece of malicious software used by a hacker? Some form of authentication should be used to ensure security.

3. Agent Service

Running the IDAgent as an application under Java required a few work-arounds during testing. If the IDAgent was running and the user logged off, the IDAgent would terminate leaving no protection. The answer to this problem is to run IDAgent as an NT service. This was done successfully; however, the user interface cannot be seen or accessed making it difficult to monitor the agent. These problems would have to be overcome to successfully use the agent in a live network.

4. Using Another Programming Language

The version of Java used in this implementation is an interpreted language and as such runs much slower than an application written in a lower level language. Java was sufficient for prototyping and allowed rapid development of the communication portions of the agent. However, other languages should be researched.

5. Response to Attack

There must be a response to an attack or intrusion to prevent entry. The system should be reactive.

6. Sensors

The IDAgent tested here had limited sensor capability. It could detect user login attempts and when another agent was not responding. Other sensors could scan for network traffic patterns, known attacks, or other system log entries. The agent was written in a modular fashion to allow such sensor threads to be included easily.

7. Threshold Values

The threshold values in the agent were set based on my knowledge of network administration. Testing on a live network would allow the adjustment of the threshold values to better match the nature of the users in the network.

8. Configuration File

A configuration file that would allow an administrator to change parameters, variables, and threshold values without modification of the IDAgent would be a beneficial addition to the system.

APPENDIX A: PROGRAM CODE

```
/**
□
 * IDAgent is the main executable class of the java program.  All global variables
□
 * are defined here and all internal threads for other classes are initialized
□
 * and started here.  The Agent is designed to run independently on a computer
 * transmitting and receiving messages to other known Agents as detections or
 * anomalies are sensed.
 *
 * @author Capt Dennis Ingram, USMC
 * @version Last Updated on, %G%
 * @since JDK1.1
 * @param portnum   The port that the machine will be listening on
 *                                This is set at 8000 by default but may be changed
 *                                by the user at startup.  It must be the same for
all agents
 *                                In your network.
 *
 */

import java.io.*;
import java.net.*;
import java.lang.*;
import java.util.*;
import java.awt.*;
import java.awt.event.*;
import java.text.*;

public class IDAgent {

       final static int TRANSMITINTERVAL = 300000; //5 minutes default transmit
interval
//     static int TRANSMITINTERVAL = 60000; //1 minutes
//     static int TRANSMITINTERVAL = 30000;  //30 sec
       final static double DEGREDATION          = 0.9; // for degrading the alert
level.
       final static int HOUR                    = 3600000; // 1 hour
       final static int MINUTE                  = 60000; //1 minute
       final static String BROADCAST = "255.255.255.255";
       static double AlertLevel = 0.1; //AlertLevel between 0.0 and 1.0
       static int TotalKnownAgents = 1; //How many Agents have been on the net
count myself.
       static int TotalAlertMsgsRcvd = 0;
       static int TotalAlertMsgsSent = 0;
       static int TotalBcastRcvd = 0;
       static int TotalBcastSent = 0;
       static Date StartTime = new Date();
       static Date EndTime = new Date();

       // msgs coming in from other agents
       static Vector MsgInQueue = new Vector();

       // msgs going out to other agents
       static Vector MsgOutQueue = new Vector();

       // Internal msgs from sensors to the controller
       static Vector ControllerQueue = new Vector();
```

```
        //create a list of Known Agents.
        static Vector AgentList = new Vector();

        static AgentWindowMgr WinMgr;
        static agent_controller Controller;
        static TCPtransmit sender;

        //List of all local addresses
        static InetAddress[] LOCALADDRESSES;
        static int TCPport = 8002;

        public static void main(String args[]) throws Exception {
         //parameters = portnumber to listen on
         //example: c:\java IDAgent [portnum]  (*portnum optional).
                int portnum = 0;
                try {
                        LOCALADDRESSES =
InetAddress.getAllByName(InetAddress.getLocalHost().getHostName());
                        portnum = Integer.parseInt(args[0]); //get a port for
UDPreceiver
                }
                catch (java.net.UnknownHostException e) {}
                catch (java.lang.ArrayIndexOutOfBoundsException e) {
                        portnum = 8000; // if not set, use default 8000
                }

                try {

                        WinMgr = new AgentWindowMgr(portnum);
                        WinMgr.addWindowListener(new CloseWindowandExit());
                        Controller = new agent_controller();
                        TCPreceiver listener = new TCPreceiver(portnum);

                        System.out.println("controller starting...");
                        UDPreceiver l1 = new UDPreceiver(portnum);
                        UDPtransmit t1 = new UDPtransmit(portnum);
                        System.out.println("receiver starting...");
                sender = new TCPtransmit();
                System.out.println("transmitter starting");
                HostSensor hs1 = new HostSensor(); //checks Status of known hosts.
                AlertParser ap1 = new AlertParser(); //checks Status of old Alerts.
                        LogSensor ls1 = new LogSensor(); //check the sys log.
                }
                catch (Exception e) {System.out.println("Main exception... " + e);
                e.printStackTrace();}
        } //end main
} //end class IDAgent

/**
 * A thread within the Agent that controls the flow of message traffic and contains
 * the logic for handling incoming and sensor data.  This class is the brain
 * of the Agent.  It continually monitors incoming messages from other Agents, and
 * messages from sensor threads and takes any necessary actions to notify the user
 * and other Agents in the network of anomalies.  When there is no data to process
it
 * suspends until needed.
 *
 * @author   Capt Dennis Ingram, USMC
 *
 */

class agent_controller extends Thread {
        final double MAXALERT = 1.0;
        final double HOSTTHRESHOLD = .35;
        final int MSGTHRESHOLD = 25;
        final double NETLOGINFAILTHRESHOLD = .15;
        final double HOSTLOGINFAILTHRESHOLD = .15;
        Date LastAlertTime = new Date();
        //Vector for storing alerted msgs for use only by the controller.
        private Vector RecentAlerts = new Vector(); //within 12 hours old
        private Vector OlderAlerts = new Vector(); //over 12 hours old,purged daily
        private boolean StopSending = false; //flag on message sending.
        private int ResponseCnt = 0;

                agent_controller() {
                this.start();
        } //end agent_controller constructor

        public void run() {
        boolean run_flag = true;
        try {
                while(run_flag) {
                        if((System.currentTimeMillis() - LastAlertTime.getTime()) >
                         (IDAgent.TRANSMITINTERVAL * 2)) {
                                StopSending = false; //reset the flag if no alerts
generated.
                                ResponseCnt = 0; //reset counter.
                                //reduce alert level by Degredation factor
                                IDAgent.AlertLevel=(IDAgent.AlertLevel *
IDAgent.DEGREDATION);
                                if(IDAgent.AlertLevel < 0.05)
                                        IDAgent.AlertLevel = 0.05; //never go less than
0.05
                                LastAlertTime = new Date(); //when LastAlertTime is
reached, reset.
                        } //end if
                        //check the alert status and change the window.
                        double alrt = IDAgent.AlertLevel;
                        if(alrt > 0.0 && alrt <= 0.4) {
                                IDAgent.WinMgr.buttonPanel.setBackground(Color.green);
                                IDAgent.WinMgr.buttonPanel.repaint();
                        } //end if
                        if(alrt > 0.4 && alrt <= 0.7) {
                                IDAgent.WinMgr.buttonPanel.setBackground(Color.yellow);
                                IDAgent.WinMgr.buttonPanel.repaint();
                        } //end if
                        if(alrt > 0.7 && alrt <= 1.0) {
                                IDAgent.WinMgr.buttonPanel.setBackground(Color.red);
                                IDAgent.WinMgr.buttonPanel.repaint();
                        } //end if
                        //If nothing to do now, suspend and wait for something to do.
                        if(IDAgent.MsgInQueue.isEmpty() &&
IDAgent.ControllerQueue.isEmpty()) {
                                currentThread().suspend(); //wait until called for.
                                IDAgent.sender.resume();
                        } //end if

                        synchronized(IDAgent.MsgInQueue) {
                                if(!IDAgent.MsgInQueue.isEmpty()) { //get a message for
processing.
                                        Message inbound1 =
(Message)IDAgent.MsgInQueue.firstElement();

        IDAgent.MsgInQueue.removeElement((Message)IDAgent.MsgInQueue
                                                .firstElement());
```

```
                                  ProcessExt(inbound1);
                           } //end if
                    } //end synch MsgInQueue
                    synchronized(IDAgent.ControllerQueue) {
                           if(!IDAgent.ControllerQueue.isEmpty()) { //get a message
for processing.
                                  Message localmsg1 =
(Message)IDAgent.ControllerQueue.firstElement();

       IDAgent.ControllerQueue.removeElement((Message)IDAgent.ControllerQueue
                                        .firstElement());
                                  ProcessInt(localmsg1);
                           } //end if
                    } /*end synch CQ*/

              } //end while runflag -- main controller loop
       } // end try
       catch (Exception e) {System.out.println("Controller exception..  " + e);}
       } //end run

       /**
        * Processes all messages from an external source eg. another Agent.
        * @params inbound The reference to the message to process.
        */
       private void ProcessExt(Message inbound) {
              LastAlertTime = new Date(); //update at each event.
              System.out.println("Processing message from another host " + inbound
+ ":"
                     + inbound.getWeight());

              synchronized(RecentAlerts) {
                     RecentAlerts.addElement(inbound);
              } //end synch
              int code1 = inbound.getCode();
              if(code1 == 529 || code1 == 531) {
                     ResponseCnt++;
                     inbound.setWeight(0.0); //reset the weight until evaluated by
this agent.
                     CheckLogins(inbound);
              }
              //Algorithm for increasing alert level.
              //result = ((1.0 - a) * x) + a;
              IDAgent.AlertLevel = (((MAXALERT - IDAgent.AlertLevel) *
inbound.getWeight())
                     + IDAgent.AlertLevel);
       } //end method ProcessExt

       /**
        * Processes all messages or events received from internal sensors
        * eg. a failed logon attempt on the local machine.
        * @params localmsg The reference to the message to process.
        */
       private void ProcessInt(Message localmsg) {
              //only internal alerts should generate output Messages for other
Hosts.
              LastAlertTime = new Date(); //update at each event.
              System.out.println("Processing a local message " + localmsg + ":"
                     + localmsg.getWeight());
              synchronized(RecentAlerts) {
                     RecentAlerts.addElement(localmsg);
              } //end synch
              if(localmsg.getCode() == 529 || localmsg.getCode() == 531) {
                     //Failed login detected, check for other failures.
```

```
                        //if successful login detected, put in recent alerts.
                        ResponseCnt++;
                        CheckLogins(localmsg);
                }// end if
                if(localmsg.getCode() == 6) {
                        ResponseCnt++;
                        double s1 = CheckHostStatus();
                        System.out.println("Agents down = " + s1);
                        if(s1 > HOSTTHRESHOLD) {
                                localmsg.setWeight(s1);
                        } // end if s1;
                } // end if code == 6;
                System.out.println("Alert level is " + IDAgent.AlertLevel);
                IDAgent.AlertLevel = (((MAXALERT - IDAgent.AlertLevel) *
localmsg.getWeight())
                         + IDAgent.AlertLevel);
                if(ResponseCnt > MSGTHRESHOLD) {
                        StopSending = true;
                        System.out.println("Stopped Sending messages temporarily");
                        //Premise is that if an agent is sending more than a certian
number of
                        //messages in one TransmitInterval, then something is wrong. So
stop
                        //flooding the network.
                } //end if
                 System.out.println("Changing Alert level by " +
localmsg.getWeight());
                 System.out.println("New Alert level is " + IDAgent.AlertLevel);
                if(localmsg.getCode() != 6 && !StopSending) {
                        //if msg is 'host not responding', don't send msg to others.
                        //if StopSending flag set, don't send anymore messages right
now.
                        synchronized(IDAgent.MsgOutQueue) {
                                IDAgent.MsgOutQueue.addElement(localmsg);
                                IDAgent.sender.resume();
                        } // end synch
                } // end if
        } //end method ProcessInt

        /**
         * Processes the RecentAlerts list and purges the old contents to
         * a new list called OlderAlerts.
         * This assists in the determination of whether an Alert has already come
         * in recently on a particular problem.
         * It also purges any messages older than 24 hours from the OlderAlerts
list.
         */
        public void CheckForOldAlerts() {
                System.out.println("Checking for old Alerts ");
                if(!OlderAlerts.isEmpty()) { //purge old messages
                        Message m1 = new Message(); // allocate m1
                        for (Enumeration e = OlderAlerts.elements();
e.hasMoreElements();){
                                m1 = (Message)e.nextElement();
                                if((System.currentTimeMillis() -
m1.getTimeStamp().getTime()) >
                                IDAgent.HOUR * 24) { //greater than 24 hours old
                                        synchronized(OlderAlerts) {
                                                OlderAlerts.removeElement(m1);
                                        } //end synch
                                } // end if
                        } //end for
                } // end if
                if(!RecentAlerts.isEmpty()) { // move non recent messages
                        Message ml = new Message(); //allocate ml
                        for (Enumeration e = RecentAlerts.elements();
e.hasMoreElements();){
                                ml = (Message)e.nextElement();
                                if((System.currentTimeMillis() -
ml.getTimeStamp().getTime()) >
                                (IDAgent.HOUR * 12)) { // move message to older Alerts
list.
                                        synchronized(RecentAlerts) {
                                                OlderAlerts.addElement(ml);
                                                RecentAlerts.removeElement(ml);
                                        } //end synch
                                } // end if
                        }// end for
                } // end if
        } //end method CheckForOldAlerts()

        /**
         * This method is used when displaying the relevant alerts on the screen
         * for the user.
         * @params code            A code used for debugging.  Under normal
circumstances
         *                                 only relevant messages are displayed.  If
the code is
         *                                 set to 1 then all messages will be
displayed for debugging
         */
        public void ShowAlerts(int code) {
                Message ml = new Message();
                IDAgent.WinMgr.display("There are " + RecentAlerts.size()
                        + " Recent Alerts and " + OlderAlerts.size() + " Old Alerts");
                IDAgent.WinMgr.display("Displaying relevant messages");
                for (Enumeration e = RecentAlerts.elements(); e.hasMoreElements();){
                        ml = (Message)e.nextElement();
                        if(ml.getWeight() > 0.0 && code == 0)
                                IDAgent.WinMgr.display(ml.toString());// print selected
                        else
                                System.out.println(ml.toString()); //print all

                }// end for
                System.out.println("There are " + OlderAlerts.size() + " Old
Alerts");
                System.out.println("Displaying messages");
                for (Enumeration e = OlderAlerts.elements(); e.hasMoreElements();){
                        ml = (Message)e.nextElement();
                        if(ml.getWeight() > 0.0 || code == 1)
                                System.out.println(ml.toString()); //code 1 = print all.
                }// end for

        }// end method ShowAlerts()

        /**
         * This method checks the status of hosts when a host not responding message
         * is found.  If the number of hosts down is > the HOSTTHRESHOLD value then
         * increase the alert level according to the percentage of down hosts.
         * It only runs a check when an alert to a down host is received.
         * @returns double         The percentage of agents that are down.
         * @author Dennis J Ingram, USMC.
         */
        public double CheckHostStatus() {
        int cntr = 0;
```

```
            Message m1 = new Message(); //allocate mem for a new message.
            for (Enumeration e = RecentAlerts.elements(); e.hasMoreElements();){
                  m1 = (Message)e.nextElement();
                  if(m1.getCode() == 6) {
                        cntr++;
                        System.out.println("Check Host -- Found 1 " +
m1.getWeight());
                        if(m1.getWeight() > 0.0) {
                              cntr = 0; //reset, all previous already used.
                        } // end if wt.
                  } // end if code.
            }// end for
            if(cntr > 0) {
                  //I have an alert I am processing, if there are more, use this
one also.
                  cntr++;
            }
            System.out.println("cntr is " + cntr + " Total agents is " +
IDAgent.TotalKnownAgents);
            return (cntr/(double)IDAgent.TotalKnownAgents); //return the
percentage of down agents.
      }// end method CheckHostStatus.


      /**
       * This method performs all necessary checks on previous login attempts
       * to try to determine a pattern of intrusion.  If a pattern is detected
       * on the current host or across multiple hosts, then a message is sent to
       * all other hosts to warn of the intrusion.
       * @params m1  The local message that was generated.
       * @author Dennis J Ingram, USMC
       */
      public void CheckLogins(Message in1) {
            boolean found = false;
            double TotalNetLoginAttempts = 0.0;
            double TotalHostLoginAttempts = 0.0;
            double NetLoginFailures = 0.0;
            double HostLoginFailures = 0.0;
            double PercentNetLoginFailures = 0.0;
            double PercentHostLoginFailures = 0.0;
            Vector LoginTable = new Vector(); //Temp Vector of User Login
Failures.
            Message m1 = new Message(); //allocate mem for a new message.

            for (Enumeration e = RecentAlerts.elements(); e.hasMoreElements();){
                  m1 = (Message)e.nextElement();
                  int code = m1.getCode();
                  boolean local = false;
                  for(int i=0;i < IDAgent.LOCALADDRESSES.length;i++) {
                        if(m1.getFromAddress().getHostAddress().equals
                        (IDAgent.LOCALADDRESSES[i].getHostAddress())) {
                              local = true;
                        } // end if
                  }//end for

                  switch(code) {
                        case 0   : //A locally generated code.
                              //reset all counters
                              TotalNetLoginAttempts = 0.0;
                              TotalHostLoginAttempts = 0.0;
                              NetLoginFailures = 0.0;
                              HostLoginFailures = 0.0;
```

```
                                  break;
                           case 528 : //Successful login
                                  if(local) {
                                         System.out.println("Code 528 local");
                                         TotalHostLoginAttempts++;
                                         TotalNetLoginAttempts++;
                                  } //end if
                                  else {
                                         TotalNetLoginAttempts++;
                                  } //end else
                                  break;
                           case 529 : //Failed Login
                                  if(local) {
                                         System.out.println("Code 529 local");
                                         TotalHostLoginAttempts++;
                                         TotalNetLoginAttempts++;
                                         HostLoginFailures++;
                                         NetLoginFailures++;
                                  } //end if
                                  else {
                                         NetLoginFailures++;
                                         TotalNetLoginAttempts++;
                                  } //end else
                                  found = false;
                                  for(Enumeration e1 = LoginTable.elements();
e1.hasMoreElements();) {
                                         LoginRecord r1 =
(LoginRecord)e1.nextElement();
                                         if(r1.Name.equals(m1.getID())) {
                                                r1.Fail++;

       r1.Machines.addElement(m1.getFrom()); //where attempt came from
                                                found = true;
                                         } // end if
                                  }//end for
                                  if(!found) {
                                         LoginRecord L2 = new LoginRecord();
                                         L2.Name = m1.getID();
                                         L2.Fail++;
                                         L2.Machines.addElement(m1.getFrom());
//where attemp came from.
                                         LoginTable.addElement(L2);
                                  }//end if (!found)
                                  break;
                           case 531 : //Acct Locked, Failed Login
                                  found = false;
                                  if(local) {
                                         System.out.println("Code 531 local");
                                         TotalHostLoginAttempts++;
                                         TotalNetLoginAttempts++;
                                         HostLoginFailures++;
                                         NetLoginFailures++;
                                  } //end if
                                  else {
                                         NetLoginFailures++;
                                         TotalNetLoginAttempts++;
                                  } //end else
                                  for(Enumeration e1 = LoginTable.elements();
e1.hasMoreElements();) {
                                         LoginRecord r1 =
(LoginRecord)e1.nextElement();
                                         if(r1.Name.equals(m1.getID())) {
                                                r1.Fail++;
```

```
	r1.Machines.addElement(m1.getFrom());
							found = true;
						} // end if
					}//end for
					if(!found) {
						LoginRecord L2 = new LoginRecord();
						L2.Name = m1.getID();
						L2.Fail++;
						L2.Machines.addElement(m1.getFrom());
						LoginTable.addElement(L2);
					}//end if (!found)
					break;
				case 538 : //Successful Logoff
						//No action taken
					break;
				default  :
					break;
			}// end switch()
		}// end for
		PercentNetLoginFailures = (NetLoginFailures/TotalNetLoginAttempts);
		if(PercentNetLoginFailures-(1/TotalNetLoginAttempts) >
NETLOGINFAILTHRESHOLD) {
			in1.setWeight((PercentNetLoginFailures-
(1/TotalNetLoginAttempts))
				- NETLOGINFAILTHRESHOLD);
			System.out.println("New wt set Net%=" + PercentNetLoginFailures
+ " wt=" +
				in1.getWeight());
		}
		PercentHostLoginFailures =
(HostLoginFailures/TotalHostLoginAttempts);
		if((PercentHostLoginFailures-(1/TotalHostLoginAttempts) >
HOSTLOGINFAILTHRESHOLD)
			&& ((PercentHostLoginFailures-(1/TotalHostLoginAttempts))
				- HOSTLOGINFAILTHRESHOLD > in1.getWeight())) {
			in1.setWeight((PercentHostLoginFailures-
(1/TotalHostLoginAttempts))
				- HOSTLOGINFAILTHRESHOLD);
			System.out.println("New wt set Host%=" +
PercentHostLoginFailures + " wt=" +
				in1.getWeight());
		}
		System.out.println("Total Login Attempts " + TotalNetLoginAttempts);
		System.out.println("Total Attempts this host " +
TotalHostLoginAttempts);
		System.out.println("Total Failed Logins " + NetLoginFailures);
		System.out.println("Total Failures this Host " + HostLoginFailures);
		// enumerate LoginTable and check for trends here.
		// Future work.
	}//end method CheckLogins.

	/**
	 * This method defines the response that the controller will take to a
specified
	 * attack or alert.  A response may range from turning on more enhanced
monitoring
	 * to shutting down the network interface for a period of time.
	 *
	 * @params response        The response code representing the action to be
taken.
	 * @author Capt Dennis Ingram, USMC
	 */
```

```
       public void Responder(int response) {
              switch (response) {
                     default :
                            break;
              }// end switch
       } // end method
} //end class

/**
 * Processes all incoming messages from other Agents and sends them to the
 * Controller for processing.  Chooses a random port to listen on and informs
 * the broadcast mechanism to inform other Agents.
 *
 * @author   Capt Dennis Ingram, USMC
 * @param portnum The port number that this TCPreceiver is to listen on.
 */
class TCPreceiver extends Thread {
       int portnum;
       ServerSocket listen_socket;

   TCPreceiver(int recport) {
              portnum = recport+2;
              this.start();
       }

   public void run() {
       boolean badport;

       do {
              badport = false;
              try {
                     do {
                            IDAgent.TCPport = (int)(Math.random() * 10000);
                     } while (IDAgent.TCPport < 1025);
                     listen_socket = new ServerSocket(IDAgent.TCPport,20);
              } catch ( BindException e) { badport = true;
                            System.out.println("BadPort " + IDAgent.TCPport); }
                catch ( IOException e) { System.out.println (e);}

       } while (badport);
              try {

                     while (true) {

                            Socket sock = listen_socket.accept();
                            ObjectInputStream is = new
ObjectInputStream(sock.getInputStream());
                            try {
                                   while(true) {
                                          Message c = new Message();
                                          c = (Message)is.readObject();
                                          synchronized(IDAgent.MsgInQueue) {
                                   IDAgent.MsgInQueue.addElement((Message)c);
                                                 IDAgent.WinMgr.display("Rcvr Msg : "
+ (Message)c + ":"
                                                        + sock.getPort());
                                                 IDAgent.TotalAlertMsgsRcvd++;
                                          } // end synch
                                   } //end while
                            } //end try
                            catch (java.io.EOFException ie) { }
                            catch (java.net.NoRouteToHostException ie) {
                                    System.out.println("Computer Not Available " +
InetAddress.getLocalHost());
                              }
                              is.close();
                              sock.close();
                              IDAgent.Controller.resume();
                        } // end While(true)
                  } //end try
                  catch (Exception e) {System.out.println("Receiver exception .. " +
e);}
   } //end run
} //end class TCPreceiver

/**
 * Receives and processes all broadcasts from other Agents.  Maintains a list
 * of Known Agents in the Local Network.
 *
 * @author Capt Dennis Ingram, USMC
 * @param portnum The port that this receiver listens on for broadcasts.
 */
class UDPreceiver extends Thread {
      int portnum;

   UDPreceiver(int recport) {
                  portnum = recport;
                  this.start();
      }

   public void run() {
                  try {
                        DatagramSocket listen_socket = new DatagramSocket(portnum);
                        System.out.println("UDP--Listening on Port: " +
listen_socket.getLocalPort());

                        while (true) {
                              byte data1[] = new byte[4];

                              DatagramPacket pkt1 = new
DatagramPacket(data1,data1.length);
                              listen_socket.receive(pkt1);
                              IDAgent.TotalBcastRcvd++;

                              ContactList c1 = new
ContactList(pkt1.getAddress(),Integer.parseInt(
                                          new String(pkt1.getData())));

                              synchronized(IDAgent.AgentList) {
                                    if(!c1.exists()) {
                                          IDAgent.AgentList.addElement(c1);
                                          IDAgent.TotalKnownAgents++; //keep track of
total.
                                    } //end if
                              } //end Synchronized

                              IDAgent.Controller.resume();
                        } // end While(true)
                  } //end try
                  catch (java.io.EOFException ie) { }
                  catch (java.net.NoRouteToHostException ie) {
                        System.out.println("Computer Not Available "); }
                  catch (java.net.UnknownHostException ie) {
                        System.out.println("Unknown Host rec " + ie); }
                  catch (Exception e) {
                    System.out.println("UDPReceiver exception .. " + e);}
   } //end run
} //end class UDPreceiver

/**
 * Responsible for transmitting all message traffic to other Agents in the network.
 * Suspends until the controller signals that there is traffic to send.  Processes
 * all messages and sends them to all Known Agents in the Local Network.
 *
 * @author Capt Dennis Ingram, USMC
 */
class TCPtransmit extends Thread {
   TCPtransmit() {
             this.start();
   }

   public void run() {

       Socket s = null;
       ObjectOutputStream out = null;
       try {
             while(true) {
                          currentThread().suspend(); //wait for something to do.
                          synchronized(IDAgent.MsgOutQueue) {
                          synchronized(IDAgent.AgentList) {
                             while(!IDAgent.MsgOutQueue.isEmpty() &&
!IDAgent.AgentList.isEmpty()) {
                                  for (Enumeration e =
IDAgent.AgentList.elements(); e.hasMoreElements();){
                                        ContactList cl =
(ContactList)e.nextElement();
                                        try {
                                              s = new
Socket(cl.getHost(),cl.getPort());
                                              out = new
ObjectOutputStream(s.getOutputStream());
                                              IDAgent.TotalAlertMsgsSent++;
                                              for (Enumeration m =
IDAgent.MsgOutQueue.elements();
                                                    m.hasMoreElements() ;) {
                                                    Message temp1 = new
Message();
                                                    temp1 =
(Message)m.nextElement();

      temp1.setTarget(cl.getHost().getHostName());
                                                    temp1.setFrom();
                                                    out.writeObject(temp1);
                                                    out.flush();
                                              } //end for  more Messages
                                        out.close();
                                        s.close();
                                        } // end try
                                        catch (SocketException e1) {
                                              System.out.println("Socket caught "
+ e1);}
                      } //end for more Agents.
                      IDAgent.MsgOutQueue.removeAllElements();
                     } //end While messages is not empty
                    } //end AgentList synch
            } //end synch MsgOut

             IDAgent.Controller.resume();
```

```
              } // end while
      } // end try
          catch (Exception e) {
              System.out.println("TCPtransmit exception .. " + e);}
   } //end run TCPtransmit
} // end class TCPtransmit

/**
 * Responsible for periodically broadcasting to the network the host and port
number
 * of the Agent.  The broadcast is used by all other Agents to maintain the Known
 * Agents list and determine if an Agent is active.
 *
 * @author Capt Dennis Ingram, USMC
 * @param sendport  The port that the Agent will transmit the broadcast message to,
 *                                all agents must be listening on the same port in
the Local
 *                                Network in order to communicate.
 */
class UDPtransmit extends Thread {
              int sendport;

   UDPtransmit(int sendon) {
                     sendport = sendon;
                     this.start();
   }
   public void run() {

       try {
              InetAddress al = InetAddress.getByName(IDAgent.BROADCAST);
                     while(true) {
                            DatagramSocket sl = new DatagramSocket();
                            byte dl[] = new byte[4];
                            dl[0] = '8';
                            dl[1] = '0';
                            dl[2] = '0';
                            dl[3] = '2';
                            String s2 = String.valueOf(IDAgent.TCPport);
                            dl = s2.getBytes();
                            DatagramPacket pl = new
DatagramPacket(dl,dl.length,al,sendport);
                            sl.send(pl);
                            IDAgent.TotalBcastSent++;
                            sl.close();
                            currentThread().sleep(IDAgent.TRANSMITINTERVAL);
                     } // end while
      } // end try
              catch (SocketException el) {
              System.out.println("Socket caught " + el);}
              catch (UnknownHostException el) { System.out.println("Unknown Host
trans");}
           catch (Exception e) {
              System.out.println("TCPtransmit exception .. " + e);}
   } //end run UDPtransmit
} // end class UDPtransmit

/**
 * A class to hold login information and record counts for
 * all login attempts.  Allows for tracking of unlimited user accts.
 * This class record will only be used to hold information temporarily while
 * the alert calculations are being made.
 */
class LoginRecord {
        double Fail = 0;
        String Name = new String(); //user account used to attemp login
        Vector Machines = new Vector(); //list of machines attempted.
        /**
         * Constructor for a blank login record.
         * These records are only used temporarily in the checking of Login Failures
         */
        public LoginRecord() {}
        public String toString() {
                return (new String(Name + " Fail= " + Fail));
        } //end toString.
}//end class LoginRecord

/**
 * Provides storage fields and manipulation methods for an Agent Message, ensuring
that
 * all messages sent and received have a common format.
 * @param Code             Message code number
 * @param Data             A string data message
 * @param Identifier A string identifier of the origin of the attack if known.
 * @param Target    The target host to receive the message
 * @param From             The sending host
 * @param TimeStamp The date/time the message was constructed.
 * @param Weight    The weighted value that the message carries, the seriousness of
 *                                 the message.  Used to when calculating the
AlertLevel.
 */
class Message implements Serializable {
        private int Code;
        private String Data;
        private String Identifier;
        private InetAddress Target;
        private InetAddress From;
        private Date TimeStamp;
        private double Weight;
        //Used for the data string
        private String msglist[] = {"Unknown Attack type",
                "1 Successful Login Attempt",
                "2 Failed Login Attempt",
                "3 Invalid Login Attempt, 1 user, multi-host",
                "4 Invalid Login Attempt, multi-user, 1 host",
                "5 Invalid Login Attempt, multi-user, multi-host",
                "6 Agent Has Stopped Responding",
                "7 Test Message",
                "8 Test Message"};

        /**
         * Constructor for a Message
         */
        public Message() {
                Code = 0; // Default code
                Data = new String(msglist[Code]);
                Identifier = new String("");
                try {
                        Target = InetAddress.getLocalHost(); //return address of
localhost.
                        From = InetAddress.getLocalHost(); //Initially set both to
localhost.
                } catch (UnknownHostException e) { System.out.println("Host not
found");}
                TimeStamp = new Date();
                Weight = 0.0; //default weight.
        } // end Message default constructor
```

```
/**
 * Retrieves the Message code from a Message
 * @return Code
 */
public int getCode() {
       return Code;
} //end getCode

/**
 * Set the Code value for an existing Message, also sets the message
 * data string and the default Weight value for a message.
 * @param incode           value to set the Code to.
 */
public void setCode(int incode) {
       Code = incode;
       if(Code < msglist.length) {
              Data = msglist[Code];
              Weight = 0.0;
       }
       else {
              Data = msglist[0];
              Weight = 0.0;
       }
} //end setCode

/**
 * Gets the value of the Data String
 * @return this.Data
 */
public String getData() {
       return this.Data;
}//end getData.

/**
 * Sets the value of the Data field when different than msglist
 * @param d1 String to set data to
 */
public void setData(String d1) {
       Data = d1;
}//end setData.

/**
 * Sets the Identifier of the Message.
 * @param id The string value of the Identifier.
 */
public void setID(String id) {
       this.Identifier = id;
}//end setID.


/**
 * Gets the Identifier String of the Message.
 * @return Identifier The string value of the Identifier.
 */
public String getID() {
       return this.Identifier;
}//end getID.

/**
 * Sets the Target value of a Message
 * @param hostname  String value to set Target to
 */
```

```
public void setTarget(String hostname) {
	try {
		Target = InetAddress.getByName(hostname);
	}
	catch (Exception e) {System.out.println(e);}
} //end setTarget

/**
 * Sets the From field of a Message to the current Local Host.
 */
public void setFrom() {
	try {
		From = InetAddress.getLocalHost();
	}
	catch (Exception e) {System.out.println("setFrom exception " + e);}
} //end setFrom

/**
 * Sets the From field of a Message to a specified Host.
 */
public void setFrom(InetAddress al) {
		From = al;
} //end setFrom(InetAddress)

/**
 * Gets value of From field from a Message
 * @return String
 */
public String getFrom() {
	return this.From.toString();
} //end getFrom

/**
 * Gets InetAddress value of From field of a Message
 * @return InetAddress
 */
public InetAddress getFromAddress() {
	return this.From;
} //end getFromAddress

/**
 * Gets value of Target field from a Message
 * @return String
 */
public String getTo() { //who is the packet going to.
	return this.Target.toString();
} //end getTo

/**
 *	Gets the value of the Date field from a Message
 * @return Date;
 */
public Date getTimeStamp() {
	return this.TimeStamp;
 }

/**
 *	Sets the value of the TimeStamp to the current Time
 */
public void setTimeStamp() {
	this.TimeStamp = new Date();
} //end setTimeStamp.
```

```
	/**
	 * Sets the value of the TimeStamp to the Date/Time passed in
	 * @param d1  Date to set the time to.
	 */
	public void setTimeStamp(Date d1) {
		this.TimeStamp = d1;
	}//end setTimeStamp.

	/**
	 *	Gets the value of the Weight assigned to the current message.
	 *	@return double;
	 */
	public double getWeight() {
		return this.Weight;
	}

	/**
	 *	Sets the Weight value of the current message.
	 * @param wt  The value to set the Weight to.
	 */
	public void setWeight(double wt) {
		this.Weight = wt;
	}

	/**
	 *	Overrides Object.toString().
	 */
	public String toString() {
		String s1 = new String(Code + " " + Data + " " + getFrom()
			+ " " + TimeStamp + " " + Identifier + " " + Weight);
		return s1;
	} //end override toString
} //end class Message

/**
 * Contains the fields and methods for tracking and storing information about other
 * Known Agents in the Local Network.  This information is used when transmitting
 * messages to other hosts.
 *
 * @author Capt Dennis Ingram, USMC
 * @param a1  InetAddress of another Agent on another host computer.
 * @param Port1     The Port number of another Agent that you need to communicate
with.
 * @param ContactTime      The time stamp of the last broadcast received from an
Agent.
 */
class ContactList {
	private InetAddress a1;
	private int   Port1;
	private Date ContactTime;

	/**
	 * Constructor for a ContactList when passing in a String and a Port number
	 * @param a         A string name of the host contacting.
	 * @param p         An integer port number sent from the contacting host.
	 */
	public ContactList(String a,int p) {
		try {
			a1 = InetAddress.getByName(a);
			Port1 = p;
			ContactTime = new Date();
		} //end try
```

```
                catch (Exception e) {System.out.println("Contact List exception " +
e);}
        } //end Constructor

        /**
         * ContactList constructor passing in an InetAddress and Port number
         * @param a          An InetAddress of a Contact.
         * @param p          The port number sent by the remote host.
         */
        public ContactList(InetAddress a,int p) {
                try {
                        al = a;
                        Portl = p;
                        ContactTime = new Date();
                } //end try
                catch (Exception e) {System.out.println("Contact List exception2 " +
e);}
        } //end Constructor

        /**
         * Overrides Object.toString()
         */
        public String toString() {
                String sl = new String(al + " : " + Portl + " : " + ContactTime);
                return sl;
        } // end toString override

        /**
         * Retrieves the InetAddress of a particular Contact.
         * @return InetAddress
         */
        public InetAddress getHost() {
                return this.al;
        } //end getHost()

        /**
         * Retrieves the Port number of a particular Contact.
         * @return int
         */
        public int getPort() {
                return this.Port1;
        } //end getPort()

        /**
         * Sets the port number of the current object to a specified port number
         */
        public void setPort(int port) {
                this.Portl = port;
        } //end setPort();

        /**
         * Retrieves the time that an Agent last broadcast to the network.
         * @return Date
         */
        public Date getDate() {
                return this.ContactTime;
        } // end getDate()

        /**
         * Updates a Contacts' last broadcast time.
         */
        public void setDate() {
                ContactTime = new Date();
        } // end setDate()

        /**
         * Determines if a Contact is either already in the Known Agents list or
         * if the Agent is the Local Host computer.
         * @returns boolean
         */
        public boolean exists() {
                int nocontact = 0;
                for (Enumeration e2 = IDAgent.AgentList.elements() ;
e2.hasMoreElements() ;) {
                        ContactList c2 = (ContactList)e2.nextElement();
                        if(this.a1.equals(c2.a1)) {
                                c2.setDate(); // update the Agent.ContactTime.
                                c2.setPort(this.Port1); //update the port number
                                                         // in case the agent
was brought down and back up
                                return true;
                        } // endif
                }// end enum
                for(int i=0;i < IDAgent.LOCALADDRESSES.length;i++) {

        if(this.a1.getHostAddress().equals(IDAgent.LOCALADDRESSES[i].getHostAddress(
))) {
                                return true;
                        } //endif
                } //endfor
                return false;
        }// end exists();

} //end class ContactList

/**
 *     This class is the GUI interface that the user sees.  It contains
 *  a text area for messages and several buttons for performing debug
 *  and information retrieval.
 *     @author James Breitinger, Major, USMC.
 */
class AgentWindowMgr extends Frame implements ActionListener {
        // establish frame window area
        TextArea textArea1 = new TextArea();
        Button gettotal = new Button("Get Totals");
        Button showagent = new Button("Agents");
        Button alert = new Button("Alerts");
        Button alertlvl = new Button("Status");
        Panel buttonPanel = new Panel();
        Color backgroundColor = Color.green;

        public AgentWindowMgr(int port1)
        {
                //define window area
       super("IDAgent: " + port1);
          setSize(500,320);
       add(textArea1, BorderLayout.CENTER);
       buttonPanel.setBackground(backgroundColor);
       buttonPanel.setLayout(new FlowLayout(FlowLayout.CENTER,50,5));
       buttonPanel.add(gettotal);
       buttonPanel.add(showagent);
       buttonPanel.add(alert);
       buttonPanel.add(alertlvl);
       add(buttonPanel, BorderLayout.SOUTH);
          gettotal.addActionListener( this );
          showagent.addActionListener( this );
         alert.addActionListener(this);
         alertlvl.addActionListener(this);
      setVisible(true);
       } //end constructor

       public void actionPerformed(ActionEvent e)
       {
             Object source = e.getSource();
             boolean done = false;
             try {
                    if (source == gettotal) {
                           System.out.println("Total Msgs Recvd " +
IDAgent.TotalAlertMsgsRcvd);
                           System.out.println("Total Msgs Sent " +
IDAgent.TotalAlertMsgsSent);
                           System.out.println("Total Bcast Recvd " +
IDAgent.TotalBcastRcvd);
                           System.out.println("Total Bcast Sent " +
IDAgent.TotalBcastSent);
                           System.out.println("Start Time " + IDAgent.StartTime);
                           IDAgent.EndTime = new Date();
                           System.out.println("End Time " + IDAgent.EndTime);
                           IDAgent.Controller.ShowAlerts(1);
                    } //end if submitmsg
                    if (source == showagent) {
                           display("There are " + IDAgent.AgentList.size() + "
known Agents.");
                           for (Enumeration e2 = IDAgent.AgentList.elements();
e2.hasMoreElements();){
                                 ContactList cl = (ContactList)e2.nextElement();
                                 display(cl.toString());
                           }// end enum
                           IDAgent.Controller.resume();
                    } // end if showagents
                    if(source == alert) {
                           IDAgent.Controller.ShowAlerts(0);
                    } //end if
                    if(source == alertlvl) {
                           display("Current Alert Level is : " +
IDAgent.AlertLevel);
                    } //end if
             } //end try
             catch (Exception se) { System.out.println("Action exception " + se);}
       } //end actionPerformed
       /**
        * Method called to display a string in the text area of the Window Manager.
        */
       public void display(String s) {
             textArea1.append( s + "\n");
       } //end display
} //end class AgentWindowMgr

/**
 * Remote Host sensor that detects when a host drops off line.
 * It will send an alert message to the current host controller
 * for each host that is down.
 * @author  Dennis J. Ingram USMC
 */
class HostSensor extends Thread {

       HostSensor() {
             setDaemon(true);
             this.start();
	} // end Constructor

	public void run() {

		while(true) {
			System.out.println("Host Sensor Running...");
			Message ml = new Message();
			for (Enumeration e2 = IDAgent.AgentList.elements();
e2.hasMoreElements();){
				ContactList cl = (ContactList)e2.nextElement();
				if((System.currentTimeMillis() - cl.getDate().getTime())
>
				(IDAgent.TRANSMITINTERVAL * 3.5)) {
					System.out.println("System down " + cl);
					ml.setCode(6);
					ml.setFrom(cl.getHost());
					synchronized(IDAgent.ControllerQueue) {
						IDAgent.ControllerQueue.addElement(ml);
					} //end synch
					IDAgent.AgentList.removeElement(cl); // remove
from list
				}//end if
			}// end for
			try {
				IDAgent.Controller.resume();
				currentThread().sleep(IDAgent.TRANSMITINTERVAL);
			} catch (java.lang.InterruptedException e) {
					System.out.println(e);}
		} // end while
	} // end run
}//end class HostSensor

/**
 * This Thread runs periodically (every 6th broadcast interval) to
 * scan the RecentAlerts list in the Agent_Controller for old msgs.
 * If the messages are over 12 hours old, they are moved to the OlderAlerts list
 * and are no longer processed under normal conditions.
 * @author  Dennis J. Ingram USMC
 */
class AlertParser extends Thread {

	AlertParser() {
		setDaemon(true);
		this.start();
	} // end Constructor

	public void run() {
		while(true) {
			try {
				System.out.println("Parser Running..." + (new Date()));
				IDAgent.Controller.CheckForOldAlerts();
				IDAgent.Controller.resume();
				currentThread().sleep(IDAgent.TRANSMITINTERVAL * 6);
			} catch (java.lang.InterruptedException e) {}
		} // end while
	} // end run
}//end class CheckAlerts

/**
 * This Thread runs periodically (every 3rd broadcast interval, 15min) to
 * scan the Security Log and detect any problems that might be occuring.
 * If something is detected, new Alert messages are generated.
 * @author  Dennis J. Ingram USMC
 */
class LogSensor extends Thread {
       Date LastProcessedTime = new Date();
       LogSensor() {
              this.start();
       } //end LogSensor

       public void run() {
              File f1 = new File("c:\\secdata.log");
              try {
                     Process myProcess = Runtime.getRuntime().exec("bu_notify.exe");

              } catch (java.io.IOException ee) {
              System.out.println("IO Error starting the notify process");}
              String s1 = new String();
              while(true) {
                     try {
                            System.out.println("Log Sensor Running...");
                            FileReader fr1 = new FileReader(f1);
                            BufferedReader br1 = new BufferedReader(fr1);
                            while(br1.ready()) {
                                   s1 = new String(br1.readLine());
                                   Logprocessing(s1); //to process the new log.
                            }// end while
                            br1.close();
                            fr1.close();
                            f1.delete();
                            LastProcessedTime = new Date(); //update the last time
read.
                            IDAgent.Controller.resume(); //start up the controller
when done.
                     } catch (java.io.FileNotFoundException e){}
                       catch (java.io.IOException e) {
                                   System.out.println("IO Exception
BufferedReader");}
                     try {
                            currentThread().sleep(IDAgent.TRANSMITINTERVAL);
                     } catch (java.lang.InterruptedException e) {}
              } // end while
       } //end run

       /**
        * Logprocessing produces messages in the message class format from the
        * strings that are read in from the log file.  Each String is processed
        * as a new message.  Each one is checked for validity and to see if it was
        * processed previously.
        * @param s1        String parameter used to parse and create a message.
        */
       protected void Logprocessing(String s1) {
              Date dt3 = new Date();
              StringTokenizer st1 = new StringTokenizer(s1,",");
              String dt1 = st1.nextToken();
              String time1 = st1.nextToken();
              int code1 = (int)Integer.parseInt(st1.nextToken());
              int code2 = (int)Integer.parseInt(st1.nextToken());
              int code3 = (int)Integer.parseInt(st1.nextToken());
              if(code1 != 528 && code1 != 529 && code1 != 531 && code1 != 538 &&
code1 != 539)
                     return; // don't process anything if the codes don't match.
              String Account = st1.nextToken(); //temp
              String temp1 = st1.nextToken(); //temp
              String SourceCmptr = st1.nextToken();
              Account = Account.toUpperCase();
              String date2 = new String(dt1 + " " + time1);
              try {
                     SimpleDateFormat df =
(SimpleDateFormat)DateFormat.getDateTimeInstance();
                     df.applyPattern("MM/dd/yy HH:mm:ss");
                     df.setLenient(true);
                     dt3 = df.parse(date2);
              }  catch (java.text.ParseException e) {
                     System.out.println("Parse Exception " + e); }
              if(dt3.getTime() >= LastProcessedTime.getTime()) {
                     Message m1 = new Message();
                     m1.setTimeStamp(dt3);
                     m1.setCode(code1);
                     if(code1 == 529 || code1 == 531)
                            m1.setData("Login Failure");
                     if(code1 == 528) {
                            m1.setData("Login Successful");
                     } // end if
                     m1.setID(new String(Account));
                     if(code1 == 528 || code1 == 529 || code1 == 531) {
                            synchronized(IDAgent.ControllerQueue) {
                                   IDAgent.ControllerQueue.addElement(m1);
                            }//end synch
                     }// end if
              }// end if date >=()
       }//end Logprocessing()
} //end class LogSensor

/**
 * Class that cleans up after a Graphics window closes
 * Taken from Dietel Java Programming examples.
 */
class CloseWindowandExit extends WindowAdapter {
   public void windowClosing( WindowEvent e )
   {
              try {
                     Process myProcess = Runtime.getRuntime().exec("kill.exe
bu_notify.exe");
              } catch (java.io.IOException e1) {}
              System.exit( 0 );
   }
} //end class CloseWindowandExit
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[BARRUS97] Barrus, Joseph D. *Intrusion Detection in Real Time in a Multi-Node Multi-Host Environment.* Master's Thesis, Naval Postgraduate School, Monterey, CA, September 1997.

[COURTIOS98] Courtios, Todd, *Java Networking & Security,* Prentice-Hall, Inc., Upper Saddle River, NJ, 1998.

[DURST99] Durst, Robert. Terrence Champion, Brian Witten, Eric Miller, and Luigi Spagnuolo, "Testing and Evaluating Computer Intrusions Detection Systems", *Communications of the ACM,* July 1999, Vol 42, No. 7, 53-61.

[HALE98] Hale, Ron, "Intrusion Crack Down", *Information Security*, August 1998.

[GAO96] Government Accounting Office Report. "Information Security: Computer Attacks at Department of Defense Pose Increasing Risks" GAO/AIMD-96-84, May 22, 1996.

[KREMER99] Steven Kremer, *Real-time Intrusion Detection for Windows NT Based on Navy IT-21 Audit Policy*, Masters Thesis, Naval Postgraduate School, Monterey, CA, 1999.

[MANASI98] Manasi, Mark, *Mastering Windows NT Server 4,* fifth edition, Sybex Inc., Alameda CA, 1998.

[NEUMANN99] Neumann, Peter, G., and Phillip A. Porras, "Experience with EMERALD to Date", Proceedings 1st USENIX Workshop on Intrusion Detection and Network Monitoring Santa Clara, CA, April 1999.

[PROCTOR96] Proctor, Paul E., "Computer Misuse Detection System (CMDS) Concepts", Science Applications International Corporation (SAIC), May 1996.

[SOBIREY99] Sobirey, Michael, and Birk Richter, "The Intrusion Detection System AID", Brandenburg University of Technology at Cottbus, http://www-rnks.informatik.tu-cottbus.de/~sobirey/aid.e.html

[ZAMBONI98] Zamboni, Diego. Jai Sundar Balasubramaniyan, Jose Omar Garcia-Fernandez, David Isacoff, Eugene Spafford, “An Architecture for Intrusion Detection Using Autonomous Agents”, COAST Technical Report 98/05, COAST Laboratory, Purdue University, June 1998.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center....................2
8725 John J. Kingman Road, Ste 0944
Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library 2
Naval Postgraduate School
411 Dyer Rd.
Monterey, CA 93943-5101

3. Director, Training and Education 1
MCCDC, Code C46
1019 Elliot Road
Quantico, VA 22134-5027

4. Director, Marine Corps Research Center 2
MCCDC, Code C40RC
2040 Broadway Street
Quantico, VA 22134-5107

5. Director, Studies and Analysis Division 1
MCCDC, Code C45
3300 Russell Road
Quantico, VA 22134-5130

6. Marine Corps Representative 1
Naval Postgraduate School
Code 037, Bldg. 234, HA-220
699 Dyer Road
Monterey, CA 93940

7. Marine Corps Tactical Systems Support Activity....................1
Technical Advisory Branch
Attn: Maj. J. C. Cummiskey
Box 555171
Camp Pendleton, CA 92055-5080

8. Chairman, Code/CS....................1
Computer Science Department
Naval Postgraduate School
Monterey, CA 93940-5000

9. Dr. Neil Rowe...1
Computer Science Department, Code CS/Rp
Naval Postgraduate School
Monterey, CA 93943-5100

10. Dr. Geoffrey Xie...1
Computer Science Department, Code CS/Xg
Naval Postgraduate School
Monterey, CA 93943-5100

11. Capt. Dennis J. Ingram..... 2
P.O. Box 988
Quantico, VA 22134-0988